Using Twinning to Adapt Programs to Alternative APIs

Marius Nita David Notkin
Computer Science & Engineering
University of Washington
{marius,notkin}@cs.washington.edu

ABSTRACT

We describe *twinning* and its applications to adapting programs to alternative APIs. Twinning is a simple technique that allows programmers to specify a class of program changes, in the form of a *mapping*, without modifying the target program directly. Using twinning, programmers can specify changes that transition a program from using one API to using an alternative API.

We describe two related mapping-based source-to-source transformations. The first applies the mapping to a program, producing a copy with the changes applied. The second generates a new API that abstracts the changes specified in the mapping. Using this API, programmers can invoke either the old (replaced) code or the new (replacement) code through a single interface.

Managing program variants usually involves heavyweight tasks that can prevent the program from compiling for extended periods of time, as well as simultaneous maintenance of multiple implementations, which can make it easy to forget to add features or to fix bugs symmetrically. Our main contribution is to show that, at least in some common cases, the heavyweight work can be reduced and symmetric maintenance can be at least encouraged, and often enforced.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.12 [**Software Engineering**]: Interoperability

Keywords

Twinning, API mapping, source-to-source translation

1. INTRODUCTION

Developers often implement and maintain code artifacts as variations of existing programs. This in turn creates the challenge of keeping the common parts of the variants up-to-date with one another while still allowing the varying parts to evolve independently.

ICSE '10, May 2-8 2010, Cape Town, South Africa

An increasingly familiar example of this arises when modifying a program to use an alternative application programming interface (API), either in place of or in addition to an existing one. Sometimes, the task is to support libraries that may be installed some places but not others (for example, graphical user interfaces), or to support libraries with different performance characteristics, etc. Other times, the programmer may want to reuse an existing function implemented by code that uses different APIs than the ones desired. For example, to create a GUI to a new online social service, it may be much more straightforward to retrofit an existing GUI for another social service than to start from scratch.

The proliferation of open source software and code search tools makes it increasingly simpler to find large sets of similar APIs and existing code. As just two among myriad examples, there are around forty XML parsing libraries and a dozen online map services, each with unique strengths and weaknesses. Tool support for transitioning code to use these APIs is largely missing, leaving programmers to perform the implementation and maintenance manually.

Additionally, effective ways to manage these activities depend, in part, on the ownership of both the original program and the variant. For example, if a developer's team owns both, then changes to code that is shared can be propagated to both implementations. In cases where the variant is a independent fork ("clone-and-own"), such changes are more difficult, perhaps reducing the approach to simply triggering some concern that corresponding changes may be warranted in the other implementation.

Regardless of the ownership model, the general absence of tool support makes it heavyweight, tedious, and error-prone to maintain the desired relationship between the common parts of the variants.

In this paper we describe *twinning*, a technique that allows programmers to maintain differences from an existing base program in terms of a code-level mapping. With twinning, we can write a mapping specifying how two APIs A and B correspond to one another, and then apply it to base code using A to get a variant of it that uses B. Ongoing changes to the base code can be reimported by applying the mapping again to a later version, possibly adjusting the mapping in the process. We also describe a transformation that uses the mapping to generate a new API C that abstracts the details of both A and B. APIs A and B can be manipulated uniformly through this abstract interface. The intent of this pair of approaches to twinning is to provide support for the varied code ownership models described above.

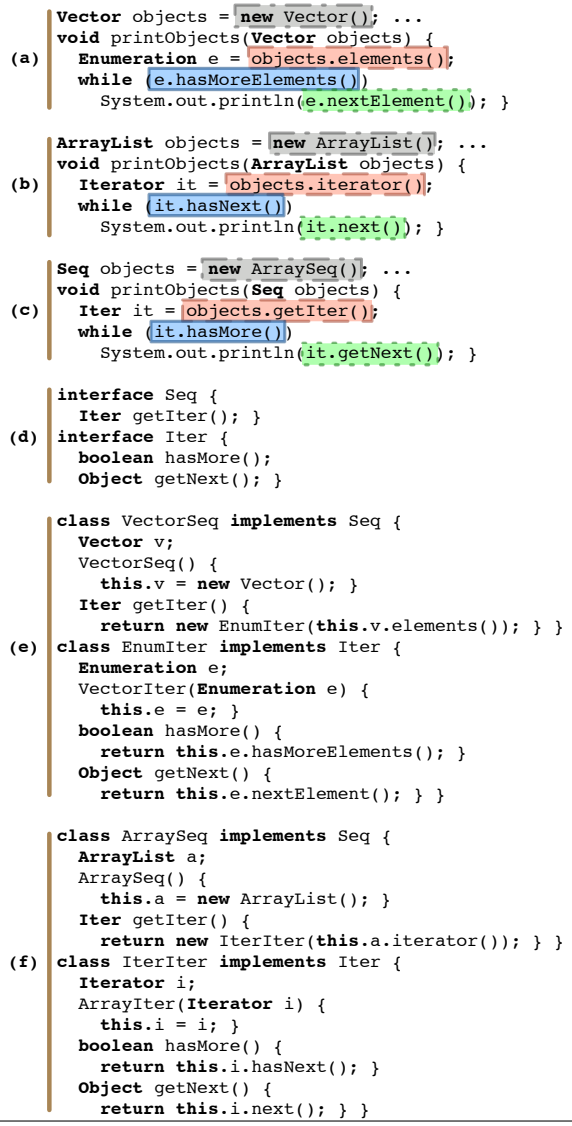

Figure 1: Two implementations of a function using similar APIs (a-b) and a third implementation (c) using an API (d-f) that abstracts both initial APIs.

The rest of the paper is structured as follows. Section 2 discusses known design patterns for managing alternatives and their tradeoffs. Sections 3 through 6 detail the core components of twinning: the mapping language and the mapping-based program transformations. Section 7 describes how we applied twinning to retrofit programs to use alternative APIs. Section 8 discusses the limitations of twinning, Section 9 describes the related work, and Section 10 concludes with a discussion of future work.

2. WHY MANUAL ADAPTATION IS HARD

There are two main approaches to modifying a program to support an API B as an alternative to an API A. The first approach modifies the subset of the program that uses API A to use API B instead. We call this approach *shallow adaptation*. The second approach, which we call *deep adaptation*, modifies the program to use a "more abstract" API C instead of A. The API C exports a set of interfaces, each of which has two underlying implementations: one using API A and one using B.

Consider the example in Figure 1. Supposing that Figure 1(a) is the original program, Figure 1(b) is the result of shallow-adapting the original program to use the `ArrayList/Iterator` API in place of `Vector/Enumeration`, where the corresponding changes are highlighted by different colors and bounding boxes. That is, the result of shallow adaptation is a copy of the program where references to one API have been replaced with references to an alternative API. Figure 1(c-f) is instead the result of deep-adapting the original to be able to use either API. Figure 1(d) is a new API that abstracts the subsets of both `Vector/Enumeration` and `ArrayList/Iterator` that are used by the program, and (e) and (f) are its two implementations. Figure 1(c) is the modification of the original that uses this new API. The result of deep adaptation is then creating the new API and its implementations, and then retrofitting the original program to use the new API.

The tradeoffs between shallow and deep adaptation are similar to the tradeoffs between code duplication and abstraction: shallow adaptation tends to be more straightforward to write but harder to maintain, whereas deep adaptation tends to be more costly to write but easier to maintain. The ownership model of the code is, again, material: deep adaptation is often infeasible—a common such situation is when the original code is on a critical path for another team.

Shallow adaptation is more straightforward to write because it changes a duplicated subset of the program, not touching and therefore not breaking any of the existing, trusted code. It is, however, harder to maintain: future additions of features, bug fixes, and tests to one copy may or may not have analogues in the other. Lacking tool support, developers can easily forget to update the copies synchronously.

Deep adaptation is easier to maintain because it clarifies the separation between API-specific code and client code: the differences are clearly delimited and tucked away from the client program, and there's only one copy of the shared code. Some concern with synchronous maintenance remains, but is reduced to the clearly delimited differences. In Figure 1(e-f), the method bodies hide only the differences between the two APIs. If a method in `EnumIter` changes, there is still the question of whether an analogous change should be made to `IterIter`.

Deep adaptation is more difficult to implement. (Recall that the situation we address is when there is already existing code for which a developer wants to create a variant. In developing from scratch, deep adaptation is more attractive from both an abstraction and a cost point of view.) First, the programmer must design a new API C. In general, if the program uses n types in API A, the programmer may have to create n new interfaces (or abstract classes) and $2 \times n$ new classes implementing those interfaces. In our example, we created two interfaces and four implementations. Second, the programmer must adapt the client program to use the new API C, changing possibly well-tested and trusted code in the process.

Finally, both shallow and deep adaptation can prevent the program from compiling for extended periods of time. Specifically, while replacing uses of an API with uses of another API — for example, consider the transition from Fig-

```
M ::= ∅                                          (blank mapping)
    | [ T ( F ) { B }                              (replacement)
        T ( F ) { B } ]
    | M ; M                                  (list of replacements)

F ::= ∅ | F, T x                                        (formals)
B ∈ Java method bodies
T ∈ Java types
x ∈ Java identifiers
```

Figure 2: Abstract syntax of the mapping.

```
[ Vector      ()                { return new Vector(); }
  ArrayList   ()                { return new ArrayList(); }   ]

[ Enumeration (Vector v)        { return v.elements(); }
  Iterator    (ArrayList a)     { return a.iterator(); }      ]

[ boolean     (Enumeration e)   { return e.hasMoreElements(); }
  boolean     (Iterator i)      { return i.hasNext(); }       ]

[ Object      (Enumeration e)   { return e.nextElement(); }
  Object      (Iterator i)      { return i.next(); }          ]
```

Figure 3: Mapping for Figure 1.

ure 1(a) to Figure 1(c) — when one declaration is changed from A to C, the program may break arbitrarily badly. The program is likely to not compile until the programmer completes the whole translation.

3. THE MAPPING

Underlying our approach is the observation that in the process of manual adaptation — either shallow or deep — the programmer implicitly specifies code-level correspondences between the APIs. For example, the programmer replaces of type `Enumeration` with type `Iterator` and of `Enumeration.nextElement()` with `Iterator. next()`.

With twinning, the developer writes these differences separately, in the form of a *mapping* whose entries specify when two different code snippets correspond to one another. From the mapping, we can automate key aspects of both shallow and deep adaptation. The mapping that helps us generate shallow and deep adapters for the code in Figure 1(a) is shown in Figure 3.

When designing the mapping representation, we balanced three inter-related principles:

1. *Ease of use.* The mapping language should resemble the programming language itself as closely as possible.

2. *Generality.* The mapping representation should allow a reasonable set of API-to-API mappings to be expressed.

3. *Tool utility.* Mappings should be easy to leverage by matching algorithms and other analyses.

As these principles stand in tension, our initial design represents a sweet spot that addressed each of them to a reasonable extent.

Figure 2 shows the abstract syntax of the mapping, which we restricted, for now, to the specification of single replacements (e.g., relating only two APIs). A mapping $\boldsymbol{M}$ is a (possibly empty) list of replacements. A replacement

```
[ T1 ( F1 ) { B1 }
  T2 ( F2 ) { B2 } ]
```

consists of two nameless procedures, each of which has a return type `T`, a list of formals `F`, and a body `B`. The replacement specifies that $\texttt{T}_1$ is related to $\texttt{T}_2$, that the list of formals $\texttt{F}_1$ is point-wise related to the list of formals $\texttt{F}_2$, and that $\texttt{B}_1$ is related to $\texttt{B}_2$. Replacements are akin to the entries of a logical relation, which relates programs that take related inputs to related outputs. In the rest of the paper, we refer to the *domain* and *range* of the mapping in the intuitive sense: above, the snippet $\texttt{B}_1$ is in the domain and $\texttt{B}_2$ in the range of the mapping.

When applying the mapping to a program (see Section 4), occurrences of $\texttt{B}_1$ are replaced by occurrences of $\texttt{B}_2$. The purpose of the return type and the list of formals is twofold. First, they are used in type-checking the mapping. Therefore, replacements must be closed programs in which all free variables in `B` are closed by `F`. Second, they can be used by mapping-based transformations. When applying a mapping to a program, for example, type matching allows identification of code to translate. Otherwise, `x.foo()` and `y.foo()` would match despite the types of `x` and `y` being different.

A *well-formed* replacement is type-correct and its argument lists are in pointwise correspondence. That is, the argument lists of the replacee and the replacer are equal in length and the ith argument in one list corresponds to the ith argument in the other. For API alternatives, this restriction means that we do not generally handle APIs where the functionality of one type in one API is spread out over the functionality of two or more types in the other. This restriction simplifies our algorithms significantly, and we have yet to find real and useful examples where removing it is absolutely necessary.

Figure 3 shows the mapping that handles the example in Figure 1. The mapping syntax does not mention the names used by the analysis that generates the adapters in Figure 1(d-f): `Seq`, `Iter`, `getIter`, `hasMore`, and `getNext`. Names can easily be incorporated by (a) attaching a name to each replacement and (b) adding a syntactic form that maps two related type names to a new name, e.g. `name Seq {ArrayList, Vector}`. Alternatively, names could potentially be automatically-generated using a heuristic algorithm that yields readable names for corresponding blocks and corresponding types.

We believe that this representation approximates our design principles. It is *natural to use*: programmers specify correspondences as pairs of Java methods. In particular, they can copy and paste snippets from the program and then specify their replacements. It is *general*: it allows code correspondences to be defined as pairs of method bodies. It is *useful*: we defined shallow and deep adaptation transformations based on it.

4. SHALLOW ADAPTATION

A program transformation

```
ShallowAdapt(P,M) = P'
```

generates a variant program `P'` from a program `P` and the mapping `M`. The transformation `ShallowAdapt` walks the program's abstract syntax tree (AST) and attempts to apply each replacement to each node. When encountering a block,

the algorithm walks over the block's statements, matching every suffix of the block, allowing sequences of statements to be replaced.

We first compute a type-mapping that gathers related type pairs:

```
TypeMapping(∅) = ∅
TypeMapping([Tr(T1 x1,...,Tn xn) { B  }
             T'r(T'1 y1,...,T'n yn) { B' }])
  = { (Tr,T'r), (T1,T'1), ..., (Tn,T'n) }
TypeMapping(M;M')
  = TypeMapping(M) ∪ TypeMapping(M')
```

For each replacement, the return types and the types of the arguments are mapped together according to their positions in the argument list. For example:

```
[ Enumeration (Vector v) { return v.elements(); }
  Iterator (ArrayList a) { return a.iterator(); } ]
```

After computing `TypeMapping(M)`, `Vector` maps to `ArrayList` and `Enumeration` to `Iterator`. An additional check verifies that the pairs form a function. That is, if `Vector` maps to `ArrayList`, then `Vector` cannot map to anything else.

When walking the AST, `ShallowAdapt` replaces each mention of type `T` (in fields, formals, and locals) with its corresponding type in the type mapping, and each matched code snippet with its replacement. The transformation does not track variable names; because the replacements have 1-1 correspondences in their argument lists, we simply re-purpose the variable names used by the expression to be replaced. For example, given the replacement above, the code

```
Enumeration elements = objects.elements();
```

is translated into

```
Iterator elements = objects.iterator();
```

In general, the way we drop the replacement code into the program is by placing it in a new scope (surrounding it by brackets) and rewriting `return` statements as assignments into the variable expecting the result of the snippet. In the common case, where a replacement is a single expression, we can replace the code in line, omitting scopes.

`ShallowAdapt` does not verify that the mapping addresses all the desired replacement sites. However, if a type-changing translation (as is the case with API-to-API mappings) misses a code snippet, the resulting program is extremely unlikely to type-check, quickly guiding the programmer toward the problem spot, after which the programmer can opt to change the program (if possible) or expand the mapping to account for the missing replacement.

4.1 Matching

The matching algorithm attempts to determine whether a replacement block `B` is equivalent to a code snippet `B'` residing within the program. Our proof-of-concept implementation performs a literal comparison on the abstract syntax of `B` and `B'` with the following exceptions:

- Types are checked. For example, `x.z()` matches `y.z()` only if the type of `x` equals the type of `y`.
- Identifier names are ignored. For example, `x.y(p)` matches `z.y(q)`.
- Occurrences of the `return` keyword in replacements are ignored in the match. For example, `{return x.foo();}` matches `x.foo()`.

The matching algorithm does not use any data- or control-flow information to handle more precise matches. Therefore, two snippets `A;B;` and `A;C;B;` are not found equivalent if `C` has no consequence on the behavior of `B`. Our simple algorithm works well for our purposes so far, mostly because replacements are small and easy to match. As discussed in Section 7, it is often possible to get around limitations in the matching algorithm by adjusting the mapping. Nothing in our approach restricts the matching algorithm, however, and much more powerful matchers can be plugged in.

5. DEEP ADAPTATION

Deep adaptation is captured by a related transformation

```
DeepAdapt(P,M) = P', A, I1, I2
```

that takes a program `P` and a mapping `M` and yields

- `A`: a set of abstract classes and interfaces, forming an API whose purpose is to abstract away the differences contained in the mapping `M`.
- `P'`: a modification of `P` that calls into the API `A` wherever `P` contains code that is in the domain of `M`.
- I_1, I_2: implementations of the API `A`. I_1 hides the implementation details of the code in the domain of the mapping and I_2 the details of code in its range.

To generate program `P'`, we will create a mapping `M'` that specifies how the domain of `M` corresponds to the new API `A`. Then, `P' = ShallowAdapt(P,M')`. To generate API `A` and its implementations, we will compute a correspondence that associates each pair of related types (as computed by `TypeMapping` in the previous section) with a set of replacements—a subset of the mapping `M`. A pair of related types represents a type in the new API `A` and its associated replacements represent its methods. Each replacement has two bodies: one body for each of the method's implementations.

We assume functions

```
ApiTypeName : T×T → String
ImplTypeName : T → String
MethodName : R → String
```

`ApiTypeName` assigns names to pairs of corresponding types. `ImplTypeName` assigns implementation-specific names to types. `MethodName` assigns names to replacements, which are ranged over by `R`. Recalling Figure 1,

```
ApiTypeName(Vector,ArrayList) = "Seq"
        ImplTypeName(Vector) = "VectorSeq"
               MethodName(r) = "getIter"
```

where `r` is the second replacement in Figure 3. That is, the name `"getIter"` names both underlying calls to `Vector.elements()` and `ArrayList.iterator()`.

5.1 Generating the API

First, we sketch how to compute the mapping that associates a pair of related types to a corresponding list of replacements:

```
RMap(∅,T,T') = ∅
RMap(R,T,T')
  = if IsConstr(R,T,T') then { C(R) }
    else if IsMethod(R,T,T') then { M(R) }
    else ∅
RMap(M;M',T,T') = RMap(M,T,T') ∪ RMap(M',T,T')
```

Given a mapping M and a pair of corresponding types T and T', the algorithm computes a set of replacements, where each replacement R is tagged either as a constructor C(R) or a method M(R). The check IsConstr returns true if R's return types are T and T', respectively, and at least one of the returned expressions is a new invocation. IsMethod returns true if T is the first argument in the domain of R and T' is the first argument in its range.

The algorithm for generating the new API A is as follows:

```
EmitApi(M) = TM := TypeMapping(M);
             for (T,T') in TM
               RM := RMap(M,T,T');
               Emit "interface" ApiName(T,T')
                    "{" EmitApiSigs(RM) "}"
```

That is, for each pair of related types, we emit a new interface whose name is given by ApiTypeName(T,T'). EmitApiSigs generates (only method) signatures using the following function, which uses ApiTypeName to generate names for related type pairs, MethodName to generate method names, and NewName to generate argument names:

```
EmitSig([T_r(T_1 x_1,...,T_n x_n) { B  }
         T'_r(T'_1 y_1,...,T'_n y_n) { B' } ] as R)
  = Emit ApiTypeName(T_r,T'_r) MethodName(R)
         "(" ApiTypeName(T_2,T'_2) NewName(), ...
             ApiTypeName(T_n,T'_n) NewName() ")" ";"
```

The argument list starts at index 2, because this is denoted by index 1 and will be held in a class field in the generated implementations.

5.2 Generating the Implementations

The algorithm for generating implementations is similar to EmitApi, except it emits a class with constructors and a field containing the encapsulated object. The algorithm is as follows:

```
EmitImpl(n,M) =
  TM := TypeMapping(M);
  for (T,T') as TP in TM
    RM := RMap(M,T,T');
    FName := NewName();
    Emit "class" ImplTypeName(sel_n(TP))
         "implements" ApiTypeName(T,T') "{"
         sel_n(TP) FName ";" // field declaration
         EmitMethods(n,RM,FName) "}"
```

EmitImpl takes a number n and a mapping M and emits a set of implementations, one for each type pair computed by TypeMapping(M). The number n is a selector index into a pair, such that sel_1(T,T') = T and sel_2(T,T') = T'. EmitImpl(1,M) thus emits a set of implementations for the domain of the mapping and EmitImpl(2,M) emits a set of implementations for its range.

The algorithm EmitMethods uses the function EmitSig to generate method signatures and a different function to emit constructor signatures. The latter is similar to EmitSig, except it uses ImplTypeName to generate the constructor return type, and it doesn't invoke MethodName. In addition, the class is given a constructor that takes a single argument of the type it encapsulates, and its body simply sets the internal field to the value of the argument.

When generating constructor bodies, every "return e;" statement is replaced with "this.field = e;" When generating method bodies, if the method's return type is in the type mapping — that is, if the return type is given by ApiTypeName(T,T') — then every "return e;" statement is replaced with

```
"return new" ImplTypeName(sel_n(T,T')) "(e);"
```

It wraps the return value with the implementation-specific class that implements the ApiTypeName(T,T') interface.

5.3 Generating the New Program

Finally, we show how we generate the new program P' that uses the newly generated API A. The key insight here is that we already have a transformation that allows us to change a program to use new APIs: ShallowAdapt. Our task is then to create a new mapping M' that specifies the correspondences between the domain of the original mapping and the new API A, and then we can generate P' by running ShallowAdapt(P,M').

The following is a sketch of the algorithm that computes the new mapping:

```
NewMapping(∅) = ∅
NewMapping([T_r(T_1 x_1,...,T_n x_n) { B  }
            T'_r(T'_1 y_1,...,T'_n y_n) { B' }] as R) =
  Ret := if T_r == "void" then "" else "return";
  Body :=
    if IsConstr(R) then
         { "return" "new" ImplTypeName(T_r)
            "(" x_1,...,x_n ")" ";" }
    else { Ret x_1 "." MethodName(R)
            "(" x_2,...,x_n ")" ";" };
  return
  [ T_r(T_1 x_1,...,T_n x_n) { B }
    ApiTypeName(T_r,T'_r)
    (ApiTypeName(T_1,T'_1) x_1,...,ApiTypeName(T_n,T'_n) x_n)
    Body ]
NewMapping(M;M') = NewMapping(M) ; NewMapping(M')
```

If the translated replacement denotes a constructor, then the generated body is a new expression, instantiating one of the underlying implementations with the given arguments. In our translation, we always pick the implementation corresponding to the domain of the original mapping M. This way, the generated program P' should behave equivalently to P. The programmer has the freedom to change these instantiation sites to use alternative implementations.

If the translated replacement denotes a method, then the generated body is a method call (in API A) on the first argument of the replacement, with the rest of the replacement arguments passed as arguments to the method.

Deep adaptation can then be defined as follows:

```
DeepAdapt(P,M) =
  M' := NewMapping(M);
  return (ShallowAdapt(P,M'), EmitApi(M),
          EmitImpl(1,M), EmitImpl(2,M));
```

Notice that deep adaptation is essentially shallow adaptation, with the extra API generation step in the middle.

6. HANDLING EXCEPTIONS

One of the biggest hurdles we encountered when designing the representation of the mapping and its associated program transformations was suitable handling of try/catch blocks. The basic issue is that two related code blocks may not only throw different exceptions, but these sets of exceptions are often asymmetric. The 1-1 mapping restriction we

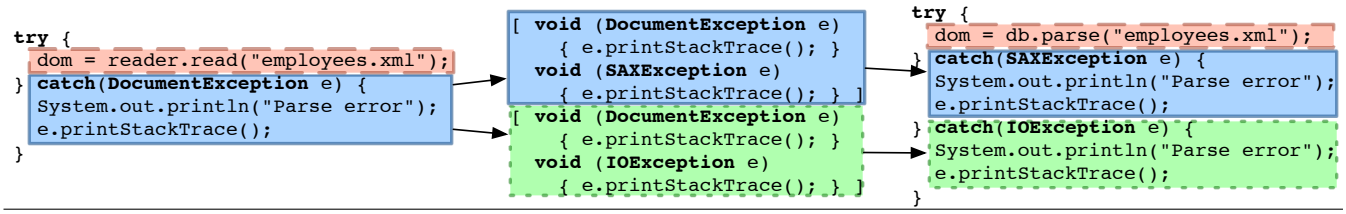

Figure 4: Using overlapping replacements to generate `catch` blocks for unevenly-mapped exceptions.

place on non-exception types would therefore be an impractical restriction on exception matching.

To handle exceptions that are thrown from a method (and hence are listed in the `throws` clause), we can simply compute the exception set of the method body *after* the translation has been applied, and place it in the `throws` clause.

To handle exceptions that are caught, we allow the mapping to contain overlapping replacements at exception types. Figure 4 shows how we map a code snippet that uses the Dom4J XML processing API to corresponding code that uses the Crimson XML API. In this example, a call to `SAXReader.read` is mapped to a call to `DocumentBuilder.parse`. `SAXReader.read` throws one exception: `DocumentException`. `DocumentBuilder.parse` throws two exceptions: `SAXException` and `IOException`. The mapping entries in the middle of the figure overlap, mapping `DocumentException` to the exceptions `SAXException` and `IOException` and specifying how it relates to each.

When translating `try/catch` statements, we need to know which `catch` blocks to translate and how, and which are extraneous and need to be deleted. We compute the `try` block's exception sets pre- and post-translation, and then consult the mapping to determine how exceptions in the two sets map together, and therefore which exceptions correspond to the statement's `catch` blocks. We process `catch` blocks as follows:

- If a `catch` block's exception does not have a correspondence in the mapping, we delete the block.
- If several `catch` blocks correspond to one exception, we pick the first (as listed in the program) and delete the rest.
- If one `catch` block corresponds to n exceptions, we replicate the block n times.

We then translate each block using its corresponding mapping entry. Figure 4, going from left to right, shows the `catch` block being replicated, and each replica being translated in the context of its corresponding mapping entry. If we were to go from right to left, we would have two `catch` blocks with only one corresponding exception, in which case we would delete the second block and translate the first.

In the generation of deep adapters, we compute exception sets directly from the mapping, and bundle them into unified exceptions. For the example above, we create the following exception classes:

```
class ParseException extends Exception { }
class Dom4JParseException extends ParseException {
  Dom4JParseException(DocumentException e) { ... } }
class CrimsonParseException extends ParseException {
  CrimsonParseException(SAXException e) { ... }
  CrimsonParseException(IOException e) { ... } }
```

Inside each implementation, when `SAXReader.read` is called, we wrap it in a `try` block whose `catch` clauses re-throw the wrapped exceptions:

```
IDocument parse(String s) throws ParseException {
  try {
    return new Dom4JDocument(this.reader.read(s)); }
  catch (DocumentException e) {
    throw new Dom4JParseException(e); } }
```

As in previous sections, a function that assigns names to related exception sets is assumed.

7. EXPERIENCE

We implemented a prototype in approximately 2,500 lines of Java code as an extension to the Cornell Polyglot source-to-source translation framework.[1] We experimented with our prototype in two separate cases. In the first case, we used it to adapt a set of small programs using the Crimson XML parsing API to use the Dom4J API instead. In the second case, we adapted a Twitter client to use the Facebook API instead.

The prototype allowed an iterative approach to defining the mappings. We inspected the code; identified a block of code to be translated; wrote the corresponding mapping entry; applied the mapping to the program; tried to compile the resulting program; and relied on type errors to identify the next snippet of code to address in the mapping. The mappings, in both cases, were less than twenty entries long.

7.1 Crimson vs. Dom4J

Using twinning, we specified correspondences between the DOM-style subsets of the Crimson and Dom4J XML parsing APIs. DOM-style XML processing retrieves an abstract syntax tree (DOM) from a parser and then traverses the tree, processing nodes.[2]

We experimented with twinning in the context of several Crimson-based example programs we found on the internet, plus the `log4j` logging API (1.2.14), which uses Crimson to process configuration files stored in XML format.

With respect to our example programs, we were able to express all the correspondences, and compile and run the resulting programs without manual modifications to the original source. Most entries are straightforward. The following entry, for example, encodes looking up an element's children, given that they have a particular name:

[1] http://www.cs.cornell.edu/projects/polyglot

[2] SAX-style parsing applies client-specified processing callbacks on the fly. As we discuss in Section 9, mapping a traversal-style parser to a callback-style one is currently outside of the range of twinning.

```
[ NodeList (org.w3c.dom.Element ele, String name)
    return ele.getElementsByTagName(name);
  List (org.dom4j.Element e, String n)
    return e.elements(n);  ]
```

Some entries required more care. Consider this program fragment:

```
DocumentBuilderFactory dbf
  = DocumentBuilderFactory.newInstance();
try { DocumentBuilder db
        = dbf.newDocumentBuilder(); ... } ...
```

The Crimson API retrieves a parser from a factory object (above), whereas the Dom4J API retrieves it directly by invoking a constructor, called `SAXReader`.[3] One sensible choice may be to write a mapping entry as follows:

```
[ DocumentBuilder () {
    DocumentBuilderFactory f
      = DocumentBuilderFactory.newInstance();
    return f.newDocumentBuilder(); }
  SAXReader () { return new SAXReader(); } ]
```

When we first did the study, our matching algorithm was not yet capable of matching across try-block boundaries. We instead created a `NoFactory` empty class and handled this case with the following two mapping entries:

```
[ DocumentBuilderFactory () {
    return DocumentBuilderFactory.newInstance(); }
  NoFactory () { return new NoFactory(); } ]
[ DocumentBuilder (DocumentBuilderFactory f)
    { return f.newDocumentBuilder(); }
  SAXReader (NoFactory f)
    { return new SAXReader(); } ]
```

Such dummies may create some confusion when reading the output code, but in terms of runtime impact, they will almost surely be eliminated by a modern optimizing compiler. Use of dummies can be reduced by an improved matching algorithm.

When generating deep adapters, we had to adjust one entry of the mapping for both translations to work properly. The original entry was as follows:

```
[ Object (NodeList l, int i) { return l.item(i); }
  Object (List l, int i)     { return l.get(i); } ]
```

Shallow adaptation works fine with this entry; post-translation, the `Object` return values are eventually cast to `org.dom4j.Element` in the program. When generating deep adapters, however, we cannot cast these return values to the abstract element type generated by our deep adaptation algorithm. We therefore had to rewrite the entry as follows:

```
[ org.w3c.dom.Element (NodeList l, int i)
    return (org.w3c.dom.Element)l.item(i);
  org.dom4j.Element (List l, int i)
    return (org.dom4j.Element)l.get(i);  ]
```

The adapter generation algorithm then generates proper method bodies, using the extra type information to encapsulate the return value into its corresponding implementation. On the Dom4J side, the algorithm generates the following method:

[3] `SAXReader` is a somewhat confusing name, in this context, for a real DOM-style parser.

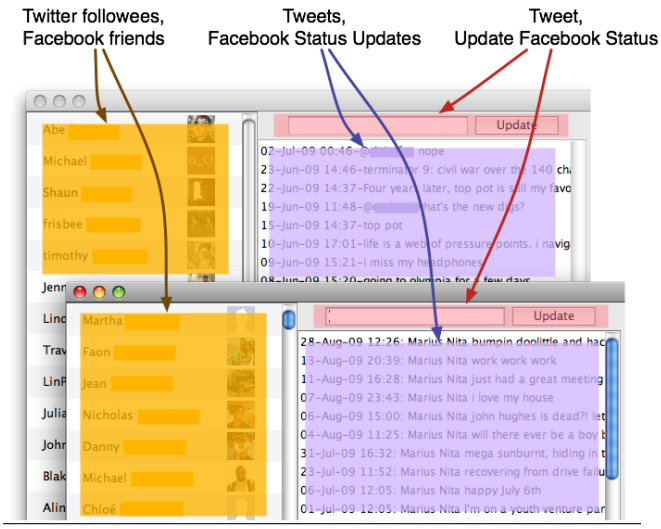

Figure 5: A partial view of `SimpleTwitter` and a modified version (in focus) that uses the Facebook API.

```
IElement getItem(int i) {
  return new Dom4JElement(
    (org.dom4j.Element)this.l.get(i)); }
```

where the enclosing class is `Dom4JList`, and `Dom4JElement` extends `IElement` (all generated by our algorithm). The field `this.l` holds the original list.

Finally, we found that Crimson vs. Dom4J contain correspondences that prevent the current implementation of twinning from working in the opposite direction. For example, Crimson's `NodeList` and `NamedNodeMap` both map to `java.util.List`. If the translations attempted to replace all relevant `List` operations with `NodeList` or `NamedNodeMap` operations, a malformed program would almost surely be generated.

7.2 Twitter vs. Facebook

We wrote a mapping that captures a set of modifications to `SimpleTwitter`, a `twitter4j`-based Twitter client, to use the Facebook API instead. Although Facebook and Twitter are services with different overall purpose and functionality, they do have similar subsets. `SimpleTwitter` displays the people you are following and their lists of tweets, and allows you to update your status, send direct messages, and search your followees' tweets. We picked this setup precisely because we wanted to apply twinning to APIs that have at least superficial differences, but may turn out to share deep structural similarities. A name-matching heuristic, for example, would likely fail when applied to these two APIs, because they have almost entirely different nomenclatures. Being able to map these two APIs together means we can reuse the vast majority of a graphical user interface that was built for an entirely different purpose, while writing only a tiny amount of GUI-related code.

Without inspecting the underlying APIs or the source code, we conjectured that we could change `SimpleTwitter` so that most or all of its functions were Facebook-based. That is, it should display your Facebook friends and their status updates, allow you to update your Facebook status, search friends' status updates, and send them private emails.

Figure 5 shows partial views of `SimpleTwitter` and `SimpleTwitter` with our mapping applied. The original list of Twitter followees is replaced with a list of Facebook friends. The list of tweets is replaced with a list of Facebook status updates, and updating the Twitter status is replaced with updating the Facebook status.

We used the Facebook API `fb4j`, the only Java Facebook API we found that made desktop integration fairly easy. We made a simple modification to `fb4j` to introduce a feature that had been present in the Facebook API but not yet rolled into `fb4j`: a method call that returns a user's list of status updates. The information was already present in the underlying XML schema; we simply enabled access to it through the `fb4j` interface. Because our version of Polyglot does not support generics, we also manually modified `SimpleTwitter` to remove the use of generics. We only deleted generics annotations and placed type casts where needed.

We systematically ported a subset of `SimpleTwitter` to use `fb4j`. An example of a simple mapping entry, specifying how to set the status in both APIs, is as follows:

```
[ void (Twitter tw, String s)
    { tw.updateStatus(s); }
  void (FacebookSession fb, String s)
    { fb.setStatus(s); } ]
```

In the process, we made use of empty dummy classes to replace whole panels in the graphical user interface with blank panels. We found this useful both for omitting features that didn't have correspondences (such as Twitter's "trends" listing), and to compile and run the resulting code mid-translation. For example, we compiled and ran a version of the program that had only translated tweets into Facebook status updates, by replacing everything else with blank panels. Dummy replacements were systematically translated into their real analogues. An example dummy replacement is as follows:

```
[ TrendsPanel (Twitter t)
    { return new TrendsPanel(t); }
  DummyTrendsPanel (FacebookSession sess)
    { return new DummyTrendsPanel(); } ]
```

`twitter4j` uses Java arrays for most of its object lists (such as the list of tweets), whereas `fb4j` uses the `List` interface. When writing the mapping entries, we made use of `Arrays.asList` to convert arrays into lists. Here's one such example:

```
[ List (Twitter tw) { return tw.getUserTimeline(); }
  List (FacebookSession fb)
    { return Arrays.asList(fb.getStatuses()); } ]
```

In creating the mapping we had to handle a call to `Twitter.getFollowing()` and its translation to Facebook's `getFriends()` method. The Facebook side of the mapping entry contains a lengthy computation over the Facebook API, including some exception handling, followed by a conversion from array to list. To handle this, we created a separate `FbUtil` class (including the method `FButil.getFriends()`) and deferred some translations to calls into this class. A better matching algorithm would avoid the need for auxiliary mappings in cases like these. However, we require the auxiliary class because our basic matching algorithm fails silently when attempting to replace a call that can occur in the middle of an expression, with a multi-statement block.

Finally, some mapping entries were resolutely program-specific. For example, one mapping entry replaces the `SimpleTwitter` login sequence — which prompts the user with a password dialog and initializes the main session object — with a Facebook analogue that involves no password dialog, but rather a background request to the Facebook servers.

8. DISCUSSION

To what degree does our approach to twinning — based on the use of mappings to drive both shallow and deep adaptation — address the problems of maintaining the common parts of program variants without compromising the ability to modify the distinct parts, especially in the context of supporting alternative APIs?

On the one hand, the jury must still be out on this question: our prototype is useful but limited, and our experience is illuminating but limited. On the other hand, the promise to improve over the current approach of handling these issues manually is clear. The mechanism of writing a mapping and using one of the transformation engines over time is likely to reduce developer effort. Furthermore, the benefits of explicitly representing the mapping — usually kept in the developer's head, if at all — can imaginably have other benefits such as helping the developer crystallize the intended relationship between the variants.

At its heart, twinning offers a way to separate a program's logic from ways in which it can depend on related APIs. This is highlighted especially well by the Twitter-Facebook example. The mappings clump together related code in the form of replacements that are largely agnostic with respect to the surrounding program structure. For example, the developer of `SimpleTwitter` can keep enhancing the UI, and we can keep applying the mapping to subsequent versions, to retrofit it into a Facebook client. For some changes to related code, adjustments must be made to the mapping: in other words, twinning attempts to separate the core computational concerns of the client from the relationship between alternative implementations. The separation of the client from an implementation is, of course, not new: it is the definition, maintenance, and use of the explicit relationship between alternative implementations that is new.

Defining the prototype and applying it to our examples deepened our understanding of the issues surrounding twinning in several dimensions, as well as helping us identify some limitations. We learned to use auxiliary and dummy classes to produce working mappings. In some cases, these auxiliary classes helped us get around limitations in the matching algorithm. In others, they helped us specify a working mapping where at first glance, the APIs did not appear to be in direct correspondence. Also, when mappings are complex, auxiliary code structures can be useful simply to make the mappings easier to understand.

Twinning works best when the differences are small. In some cases, the overall functions provided by two APIs can be similar, while their structure can be vastly different. One example is traversal- vs. callback-style parsing, where there is no straightforward structural correspondence between the parsers. Currently, we consider such API pairs to be outside the scope of this work.

Our matching algorithm requires type equality. To match all calls to a a method `T.m()` on both `T` and subtypes of `T`, we would have to write an entry for each subtype. It should be reasonably simple to extend the twinning mechanisms to

handle subtyping without the developer explicitly reasoning about the type hierarchy.

The 1-1 type mapping restriction makes our algorithms simple to understand and implement — in particular, we do not introduce new instantiation sites, we only replace existing ones — but may prevent some mappings from being expressed or cause them to be unreasonably complex.

Twinning does not yet allow expressing contextual information; for example, "match `T.m()` where `T` is the result of a call to `foo()`." It instead greedily matches all uses of `T`, which prevents writing mappings that translate only some uses of a type, but not others. In its current form, twinning is particularly well-suited for API-to-API translations, which usually involve fully removing all the uses of a type; though, as discussed in Section 7.1, this is not always the case. Loosening these restrictions is left for future work.

9. RELATED WORK

Previous work relating to our ideas can be split in three main categories: work that manages related differences between two programs, work that manages similarities, and languages that facilitate domain-specific program transformations.

Managing Related Differences. This vein of work goes back to David Parnas and his work on program families [20]. Parnas advocated techniques for managing and evolving systems as sets of related programs (families). Twinning can be viewed as a technique for managing families, by explicitly tracking the differences among members, in the domain of alternative APIs.

Aspect-Oriented Programming (AOP) [11] and its popular Java implementation AspectJ [12] were designed for handling concerns that cut across a program's usual abstraction mechanisms (classes, methods, etc.). With AOP, the programmer specifies a set of aspects separately from the program and aspects are woven into the code at the specified locations. AOP is a much more general paradigm than what we presented in this paper but we find it to be not particularly well-suited for API-to-API transitions. In particular AOP does not directly support (a) replacing sequences of statements and (b) replacing an expression with an expression of a different type. It is possible that such replacements can be encoded in AspectJ via workarounds.

ARCUM [21] is a system not unlike ours, where programmers can specify differences between two implementations in a mapping, and ARCUM generates new implementations from existing ones using the mapping. ARCUM's mapping language is more complex than ours, in two ways. First, the programmer organizes the mapping as a class hierarchy, where related code snippets implement the same interface. Second, the change description language is lower-level, specifying how to take apart and build ASTs. We believe that a class of shallow adaptations can be specified within ARCUM, although it is unclear how one would handle uneven mappings of exceptions without specifying the full AST-level details for how a particular `try` block maps to another. ARCUM doesn't support generation of deep adapters.

The work of Balaban *et al.* on class migration [1] targets migration of large code bases using outdated Java APIs to use their modern replacements. The system uses a type system and constraint solver to efficiently and precisely identify replacement sites in the face of subtle effects such as synchronization. Their translation therefore offers more precise guarantees than ours. To our understanding, the mapping language is restricted to relating API symbols that are in 1-1 correspondence in their return and argument types. While our mapping *entries* have a 1-1 restriction, we don't put restrictions on how the underlying API symbols map together. E.g., it could be that a sequence of three statements maps to a sequence of two statements. Class migration doesn't handle exceptions: it halts if the replacement code throws different exceptions than the replacee.

HULA [6] is a metaprogramming language that allows writing changes to Haskell programs separately, and applying them to the program in a type-safe manner. Semantic patches [18] extend the UNIX `patch` tool with semantic information and can be used for *collateral evolutions* [19], where one patch is applied to all the pertinent locations in a source base. Dynamic software updating [16] computes a patch from two versions of a program and can apply it to the program at runtime, without restarting the system.

SemDiff [3], RefactoringCrawler [4], and CatchUp! [9] are techniques that help the programmer retrofit a program using an outdated version of an API to use its new version. SemDiff finds code that doesn't compile and provides a list of candidate replacements. RefactoringCrawler automatically detects refactorings between two program versions. Thus, if an API change was the result of a refactoring, the crawler understands the structure of the refactoring and can use it to port the client code. CatchUp! can record refactorings on the API development side and replay them on the client's end to adjust client code to use the new API. Logical Structural Deltas [14, 13] can be used to detect and understand differences between two API versions and to change the clients accordingly.

In our own prior work [17], we designed a small extension to the C language that allows programmers to eliminate subsets of the program that are used to handle alternative data layouts (e.g., little- vs. big-endian) by using a declarative mapping language to specify how one layout relates to another. Then, the alternatives are automatically generated.

Libraries such as Mapstraction[4], Apache Commons Logging[5], and Eclipse SWT[6] are thin abstraction layers providing unified interfaces to sets of alternative APIs in the form of deep adapters. Mapstraction provides a unified interface to several map services (Google, Yahoo, OpenStreetMap, etc.), Commons Logging abstracts several logging APIs, and SWT abstracts all the GUI toolkits to which Eclipse is portable. One of our goals is to help developers quickly retrieve such uniform interfaces by specifying the differences among a set of APIs.

Managing Similarities. Projects such as Simultaneous Editing [15], Linked Editing [22], CloneTracker [5], and CReN [10] focus on management of *code clones*: program snippets with a high degree of syntactic similarity. To at least some extent, these techniques also provide ways to manage the differences between programs. If a change is made to the similar regions of a set of clones, the change is propagated to all the clones. If the change is made to a difference, the change remains local.

Twinning can be seen as managing the differences between certain types of code clones: program alternatives. In this

[4]http://www.mapstraction.com
[5]http://commons.apache.org/logging
[6]http://www.eclipse.org/swt

domain, twinning is complementary to all the systems listed above and could potentially be used to enhance code clone management. Twinning manages how the differences between clones map together, something the other systems do not capture. With twinning, a change to a difference region within a clone could be propagated to the other clones by using the mapping to determine the analogous changes.

Domain-Specific Transformations. There are several languages (e.g., SNOBOL [7], TXL [2], TAWK [8]), both general-purpose and domain-specific, that allow pattern matching over the structure of a program and can therefore be used to encode domain-specific program transformations. Twinning is related in that the mapping can be viewed as a set of $P \rightarrow P'$ pairs, where P is the pattern to match and P' is the program to drop in its place.

10. CONCLUSIONS AND FUTURE WORK

We investigated the problem of adapting programs to new APIs because creating and maintaining program alternatives is needlessly costly and difficult. We defined a simple technique called *twinning* to allow the programmer to specify a set of code mappings between alternatives, rather than making destructive program updates. We demonstrated how twinning can be used to retrofit programs to use new APIs, and in some cases, to customize those programs in a more general fashion.

An avenue for future work (beyond those mentioned in Section 8) is to devise a heuristic that generates candidate mappings relating API A to API B (a) from a program P using API A and (b) from APIs A and B. The algorithm can extract all of P's uses of A and attempt to match them to sequences of calls in B. One challenge, when extracting the uses of A from P, is to identify the smallest units of replacement. For example, `x.foo();x.bar();` may either be a whole unit that maps to corresponding code in API B, or `x.foo()` and `x.bar()` may be separate units, mapping to separate units in B.

As shown in the Twitter vs. Facebook example, mappings can be used to specify program-specific changes. We hypothesize a change-based program representation as a successor to twinning. The idea is that in addition to organizing programs as interfaces, classes, methods, etc., programmers can further structure their code with mappings. Individual mappings can specify features that can be enabled or disabled on demand. Mappings should also be composable, so that one mapping can be specified with one or more mappings as its base. This would simplify code-level management of configurations and customizations and allow changes to be specified without modifying the program, arguably reducing the potential for introducing bugs.

11. REFERENCES

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proc. OOPSLA*, 2005.

[2] J. R. Cordy, C. D. Halpern, and E. Promislow. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16, 1991.

[3] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. ICSE*, 2008.

[4] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: European Conference on Object Oriented Programming*, 2006.

[5] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. ICSE*, 2007.

[6] M. Erwig and D. Ren. A rule-based language for programming software updates. *SIGPLAN Not.*, 37(12):88–97, 2002.

[7] R. E. Griswold. *The SNOBOL4 programming language*. Bell Telephone Laboratories, 1968.

[8] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC '96: International Workshop on Program Comprehension*, 1996.

[9] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proc. ICSE*, 2005.

[10] P. Jablonski and D. Hou. Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *OOPSLA workshop on eclipse technology eXchange*, 2007.

[11] G. Kiczales and E. Hilsdale. Aspect-oriented programming. In *ESEC/FSE*, 2001.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: European Conference on Object-Oriented Programming*, 2001.

[13] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proc. ICSE*, 2009.

[14] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proc. ICSE*, 2007.

[15] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference*, 2002.

[16] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *SIGPLAN Not.*, 41(6):72–83, 2006.

[17] M. Nita and D. Grossman. Automatic transformation of bit-level C code to support multiple equivalent data layouts. In *International Conference on Compiler Construction*, 2008.

[18] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *Proc. PLOS '06*, page 10, 2006.

[19] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. *SIGOPS Oper. Syst. Rev.*, 42(4):247–260, 2008.

[20] D. L. Parnas. On the design and development of program families. *Software fundamentals: collected papers by David L. Parnas*, pages 193–213, 2001.

[21] M. Shonle, W. G. Griswold, and S. Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *ESEC/FSE*, 2007.

[22] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *IEEE Symposium on Visual Languages - Human Centric Computing*, 2004.