Crossing the Rubicon of API Migration

Ralf Lämmel[1] and Tijs van der Storm[2]

[1] Software Languages Team, Universität Koblenz-Landau, Germany
[2] SEN, CWI, Amsterdam The Netherlands

Abstract. Within the programming domain of XML processing, we set up a benchmark for API migration. The benchmark is a suite of XML processing scenarios that are implemented in terms of different XML APIs. We suggest that a relatively general technique for API migration should be capable of providing source-to-source translations between the different implementations. The benchmark involves APIs that are different enough to require more than just local rewrites for the migration. We make different attempts at API migration: wrapping, rewriting, and protocol-based translation. None of our attempts are entirely satisfactory, and we hope to provide this benchmark as a challenge to the broader programming language and automated software engineering communities.

Keywords: API, API Migration, Software Transformation, XML Processing

1 Introduction

APIs (application programming interfaces) are crucial components of our programming environments. Each API addresses some domain on a scale of "general to special" purpose. For instance, an XML API addresses the relatively general domain of XML processing. Other important and general domains (all supported by APIs) include database programming, GUI programming, and distributed programming. More specific domains (again supported by APIs) include financial exchange or version control management. Typically, multiple APIs exist for a given domain and programming environment. Also, APIs evolve over time. Further, new APIs emerge as the understanding of a domain improves and programming languages become more expressive. *What if an application wants to alter its commitment to a particular API or a particular version? This is when API migration may be needed!*

We speak of API migration when facing the following situation. Given an application that makes use of one API — the *source* API, we wish to migrate the application so that it leverages another API for the same domain instead — the *target* API. As a more concrete example, consider a Java application that uses say the JDOM API for XML programming. In an attempt to consolidate the number of "third party" APIs used in the application, the owner decides to eradicate the use of JDOM and migrate to the standardized DOM API. The owner of another application may go the exact inverse route: in an attempt to modernize the XML processing code, the owner decides to migrate from DOM to JDOM.

There are indeed multiple incentives for API migration:

new Vector()	⇒	**new** ArrayList()
int Vector:receiver.size()	⇒	**int** receiver.size()
Object Vector:receiver.firstElement()	⇒	Object receiver.get(0)
Object Vector:receiver.setElementAt(Object: val, **int**: idx)	⇒	Object receiver.set(idx, val)
void Vector:receiver.copyInto(Object: array)	⇒	**void** Util.copyInto(receiver, array)
Enumeration Vector:receiver.elements()	⇒	Iterator receiver.iterator()
new Hashtable()	⇒	**new** HashMap()
Object Hashtable:receiver.put(Object: key, Object: val)	⇒	Object receiver.put(key, val)
Enumeration Hashtable:receiver.elements()	⇒	Iterator (Collection receiver.values()).iterator()

Fig. 1. Quoted from [4]: a specification for migrating Java code to use a modern collection class. The source API is the class `Vector`. The target API is the class `ArrayList`. The specification consists of rewrite rules that apply to allocation sites and method calls in legacy code.

- Reduce the sheer number of different APIs used in the application.
- Improve intra-application coherence by using only one API for a given domain.
- Prefer a "standardized" API over a "third party" API in the application.
- Prefer a "modern" API over an "aged" API in the application.
- Commit to a new version of a given API.

Previous work on API migration Manual migration is a labor-intensive and error-prone task. Some sort of automation is clearly desirable. Research efforts have focused on the goal of making applications work with a *new version of an API*; there are two classes of approaches — both are based on API refactorings that are either recorded, or inferred, or provided as a specification. The first class of approaches replays refactorings on client source or bytecode so that it can directly use the new version of the API [20, 33, 39, 43]. The second class of approaches provides (binary) adapter layers (in the sense of the design pattern) that shields client source or bytecode from the changes of the API [10, 11, 14].

There is also previous work that goes beyond refactoring and potentially addresses incentives of API migration other than evolution along different versions of one API [8, 24, 4, 30]. Essentially such approaches involve some form of rewriting that replaces patterns over the source API by patterns over the target API . (Again, one can consider approaches at the source or the bytecode level.) The approach by Balaban, Tip, and Fuhrer [4] (see Fig. 1 for an illustration) is worth pointing out. This approach uses rewrite rules to describe the migration of allocation sites and method calls, and it uses extra checks and analyses to determine whether the rewrites can be safely applied.

Main thesis of this paper *We contend that API migration is a hard problem. In fact, we would like to call out APIs as an important form of "software asbestos" [26]. All examples of API migration that we have seen addressed in the literature essentially deal with evolution of the kind that two very similar APIs are being considered, where "very similar" may stand for "different versions" or APIs with nearly identical interfaces modulo renaming or macro expansion, with (nearly) identical observational behavior; c.f., Fig. 1. In this paper, we look into API migration when source and target APIs differ more substantially, say when local rewrite rules are insufficient to perform the migration. Instead some sort of code reorganization is needed, and the transformation must be validated and informed by an extra program analysis.*

Contributions[1]

- We present a benchmark for API migration, which we think is representative of the situation that source and target APIs differ more substantially. The example is extracted from the XML programming domain.
- We make different attempts at API migration for the benchmark. None of our attempts are entirely satisfactory, but we make an effort to understand the involved limitations, and to derive proposals for further investigation.
- We develop a grammar-based form of API protocol. Protocols can be interpreted as languages of execution traces as well as sets of control-flow graphs. Protocols give structure to specifications for API migration.

Roadmap §2 describes our benchmark for API migration. It is essentially a collection of XML processing scenarios implemented with the XML APIs DOM and JDOM. §3 attempts the reimplementation of DOM's interface in terms of JDOM and vice versa. This experiment provides a limited mapping specification between the APIs, and actually clarifies the hardness of the migration example at hand. §4 attempts a rewriting-based approach to API migration, where, however, we go beyond local rewrites. Other than that, the approach is admittedly naive, and lacks proper correctness or completeness guarantees. §5 develops a migration technique that is based on API protocols which define "disciplined" (well-understood) API-usage scenarios. Throughout the sections §3–§5 we refer to related work and thereby complement the above discussion of existing approaches to API migration. §6 concludes the paper.

2 A benchmark for API migration

Within the programming domain of XML processing, we will present and implement a few simple scenarios. We will show and discuss illustrative implementations for two XML APIs (DOM and JDOM), but we provide additional implementations online. The idea is here that an automated API migration could be expected to transform one implementation into the other (say the DOM one into the JDOM one — or v.v.) by means of a source-to-source translation.

We look at this benchmark as a "Rubicon of API migration". To paraphrase Martin Fowler [16]: *If you migrate the XML processing scenarios "Build XML document", "Query XML document", and "Update XML document" for say DOM and JDOM both ways, it probably means, you can do other forms of API migration as well.* We do not claim that our scenarios cover the XML domain in any reasonable sense. We merely assume that the scenarios and sample APIs are challenging enough to call for an API migration technique that can handle "somewhat different" APIs.

2.1 Scenario "Build XML document"

Given is a class of persons with accessors like the following:

[1] We are making the benchmark and our API migration attempts available online via the open source project SLPS: `http://slps.sourceforge.net/`. See directory `topics/apimigration`.

```
public static Document makeDocument(List<Person> contacts) {
  Document doc = new Document();
  Element root = new Element("contacts");
  document.addContent(root);
  for (Person p: contacts) {
    Element px = new Element("person");
    Element namex = new Element("name");
    namex = namex.setText(p.getName());
    px = px.addContent(namex);
    Element agex = new Element("age");
    agex = agex.setText(new Integer(p.getAge()).toString());
    px = px.addContent(agex);
    root = root.addContent(px);
  }
  return doc;
}
```

Fig. 2. Using the JDOM API, map a collection of persons to an XML document.

```
public class Person {
  public String getName() { ... }
  public int getAge() { ... }
}
```

Given a collection of persons, such as "Barack Obama" of age 47, and "John McCain" of age 72, we want to build an XML document that represents the collection. The resulting XML tree should look as follows:

```
<contacts>
  <person>
    <name>Barack Obama</name>
    <age>47</age>
  </person>
  <person>
    <name>John McCain</name>
    <age>72</age>
  </person>
</contacts>
```

Fig. 2 implements the scenario using the popular JDOM API for XML processing in Java. Documents and elements are constructed by regular constructor methods. The element constructor takes the element tag as an argument. Elements are hierarchically composed by the `addContent` method. The content of leaf elements is set by the `setText` method.

Fig. 3 implements the scenario using the standardized DOM API for XML processing. Elements are constructed by a factory method of the document. That is, element construction always refers to the hosting document. Elements are hierarchically composed by the `appendChild` method. The content of leaf elements is set by the `setTextContent` method. Before element construction can commence however, a document has to be fabricated through some stages of auxiliary factories, builders and implementations. This process can also result in a checked exception, which is accordingly caught.

```
public static Document makeDocument(List<Person> contacts) {
  DocumentBuilder b;
  try {
    DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
    b = f.newDocumentBuilder();
  } catch (ParserConfigurationException e) { ... }
  Document doc = b.getDOMImplementation().createDocument(
      null, "contacts", null);
  Element root = doc.getDocumentElement();
  for (Person p: contacts) {
    Element px = doc.createElement("person");
    Element namex = doc.createElement("name");
    namex.setTextContent(p.getName());
    px.appendChild(namex);
    Element agex = doc.createElement("age");
    agex.setTextContent(Integer.toString(p.getAge()));
    px.appendChild(agex);
    root.appendChild(px);
  }
  return document;
}
```

Fig. 3. Using the DOM API, map a collection of persons to an XML document.

2.2 Scenario "Query XML document"

Given an XML document with a collection of persons, we want to compute the average age of all persons contained in the document. To this end, we should locate all `<age>` elements, and extract their content for the aggregation of an average value.

Fig. 4 implements the scenario using the JDOM API. We use JDOM's concept of an element filter in combination with the `getDescendants` method to locate the `<age>` elements. The `getDescendants` method happens to return an *iterator* which forces us into using Java's interface for iteration, c.f., `hasNext` and `next`. The iterator does not provide a type parameter, and hence we need to use a *cast* to recover the intended type from the universal type `Object`; c.f., `(Element)....`

Fig. 5 implements the scenario using the DOM API. We locate the `<age>` elements with the help of the `getElementsByTagName` method; it queries all descendants with a given tag. The query result is a DOM-style node list which forces us into using index-based access; c.f., `item(i)`.

2.3 Scenario "Update XML document"

Given an XML document with a collection of persons, we want to increment the age value for a person on his or her birthday. More specifically, let us describe the update needed for "John McCain" on his birthday — August 29. Operationally, such an update requires iteration over all `<person>` elements, until the person of interest is found (judging by its child element for the name), and then, the content of the child element for the age is incremented accordingly.

```
public float average(Document doc) {
  int total = 0;
  int count = 0;
  ElementFilter filter = new ElementFilter("age");
  Iterator i = doc.getDescendants(filter);
  while (i.hasNext()) {
    Element e = (Element)i.next();
    total += Integer.parseInt(e.getText());
    count++;
  }
  return (float) total / (float) count;
}
```

Fig. 4. Using the JDOM API, compute average age over persons of a document.

```
public float average(Document doc) {
  int total = 0;
  int count = 0;
  NodeList nodelist = doc.getElementsByTagName("age");
  for (int i = 0; i < nodelist.getLength(); i++) {
    Element elem = (Element) nodelist.item(i);
    total += Integer.parseInt(elem.getTextContent());
    count++;
  }
  return (float) total / (float) count;
}
```

Fig. 5. Using the DOM API, compute average age over persons of a document.

Fig. 6 implements the scenario using the JDOM API. We query all `<person>` elements with the help of the `getContent` method that is parametrized by a suitable element filter very similar to the previous scenario. This query is applied to the root element rather than the document; c.f., the use of the `getRootElement` method. We use JDOM's convenient `getChild` method to retrieve the `<name>` and `<age>` children of a `<person>` element. The update of the age value boils down to the update of the text content of the `<age>` element; c.f., `getText` and `setText`.

One detail is worth mentioning. JDOM's `getContent` method for child elements returns a *list* whereas JDOM's `getDescendants` method for descendants (see the previous scenario) returns an *iterator*. These different results (for otherwise conceptually close query idioms) imply that client code has to consume the query results in different ways. API migration clearly has to cover such variation.

Fig. 7 implements the scenario using the DOM API. We end up putting to work the XPath API (i.e., `org.apache.xpath.XPathAPI`) for convenient query-based access. This additional API, which is often used in combination with the pure DOM API, compensates here for DOM's lack of a query method for child elements with a given tag. The execution of the XPath query returns a DOM-style node list that implies indexed access. The XPath API may throw a checked exception, which is accordingly

```
public void august29(Document doc) {
  ElementFilter filter = new ElementFilter("person");
  List l = doc.getRootElement().getContent(filter);
  for (Object o : l) {
    Element px = (Element)o;
    Element namex = px.getChild("name");
    Element agex = px.getChild("age");
    if (namex.getText().equals("John McCain")) {
      int age = Integer.parseInt(agex.getText());
      age++;
      agex.setText(Integer.toString(age));
      break;
    }
  }
}
```

Fig. 6. Using the JDOM API, increase the age of one person in the document.

```
public void august29(Document doc) {
 try {
  NodeList nodelist = XPathAPI.selectNodeList(doc, "/*/person");
  for (int i = 0; i < nodelist.getLength(); i++) {
   Element px = (Element)nodelist.item(i);
   Element namex = (Element)XPathAPI.selectNodeList(px, "name").item(0);
   Element agex = (Element)XPathAPI.selectNodeList(px, "age").item(0);
   if (namex.getTextContent().equals("John McCain")) {
    int age = Integer.parseInt(agex.getTextContent());
    age++;
    agex.setTextContent(Integer.toString(age));
    break;
   }
  }
 } catch (TransformerException e) { ... }
}
```

Fig. 7. Using the DOM API, increase the age of one person in the document.

caught. The update of the age value boils down to the update of the text content of the `<age>` element; c.f., `getTextContent` and `setTextContent`.

Arguably, the use of the XPath API could be avoided here. For the XML document at hand, we could be sloppy and query all kinds of elements (`<person>`, `<name>`, and `<age>`) by using DOM's `getElementsByTagName` method for all descendants with a given tag. (These elements tags do not occur in nested positions.) However, we wish to preserve the semantics of the JDOM code which filters child lists rather than all descendants. Alternatively, a pure DOM encoding could filter child lists programmatically — by plain loops. (There is a `getChildNodes` method that returns *all* child nodes.) This approach is very verbose. The involvement of the XPath API brings up the challenge of *API combinations*. That is, API migration cannot always be scoped to single APIs for the source and target of migration. Rather a combination of APIs may be faced.

3 API migration by wrapping

As a first attempt at API migration, we will use wrapping to re-implement the source API in terms of the target API. More specifically, the *interface* of the source API is implemented in terms of the target API. We will re-implement DOM via JDOM and vice versa. (We only cover the interface needed in the benchmark scenarios.) There are the following incentives for making this attempt:

– The wrapper approach can be said to provide us with a complete and type-checked mapping specification (i.e., the wrapper classes) between the two involved APIs. If we succeed to give such a specification, it may be a good baseline for other implementations of the API migration. For instance, the specification is perfectly amenable to testing because we can extend any test cases for the source API to the reimplementation.
– The wrapper approach essentially corresponds to one of the two classes of prior work on API migration that we mentioned in the introduction, i.e., the class that relies on adapter layers [10, 11, 14] — except that we do not derive the wrappers from refactorings, but we design them "from scratch".

We should clarify that we are primarily interested in source-to-source translations for API migration while wrappers do *not* actually serve such translations. However, in principle, we could attempt to combine the wrapper approach with program specialisation [35, 5, 36], and thereby install the target API in the actual application code. We will return to this option in the discussion part of the section.

3.1 Reimplementing DOM in terms of JDOM

We need to apply the adapter design pattern [17] systematically. Assuming a nearly 1:1 relationship between types of source API and target API, the reimplementation of any type of the source API is basically a wrapper/adapter type for the corresponding wrappee/adaptee type of the target API. Methods are reimplemented by delegation. There is the following correspondence of DOM and JDOM types when aiming at a reimplementation of DOM in terms of JDOM:[2]

Adapter type	**Adaptee type**
`org.w3c.dom.Document`	`org.jdom.Document`
`org.w3c.dom.Node`	`org.jdom.Content`
`org.w3c.dom.Element`	`org.jdom.Element`
`org.w3c.dom.NodeList`	`java.util.Iterator`
...	...

Fig. 8 shows some reimplemented DOM types. For instance, the `Document` type wraps a private instance of type `org.jdom.Document`. The reimplementation of

[2] The DOM API is actually *interface*-based, i.e., all the types of interest are interfaces. For simplicity, we provide a class-based wrapper implementation only; all DOM types are directly mapped to classes. A serious reimplementation must be interface-based though to be fully compatible with the DOM API, and to allow interface polymorphism, e.g., in the position of querying DOM documents by means of the XPath API; c.f., the "Update XML document" scenario.

```
public class Document {
  private org.jdom.Document doc;
  ...
  public Element getDocumentElement() {
    return new Element(doc.getRootElement());
  }
  public Element createElement(String name) {
    return new Element(new org.jdom.Element(name));
  }
  public NodeList getElementsByTagName(String string) {
    return new NodeList(
      doc.getDescendants(new ElementFilter(string)));
  }
}
public class Node {
  protected org.jdom.Content content;
  Node(org.jdom.Content node) { this.content = node; }

  // Helper needed to wrap JDOM nodes as DOM nodes
  protected static Node wrap(org.jdom.Content node) {
    if (node instanceof org.jdom.Element)
      return new Element((org.jdom.Element)node);
    else
      return new Node(node);
  }
}
public class NodeList {
  private List<Node> list;
  public NodeList(Iterator iter) {
    list = new LinkedList<Node>();
    while (iter.hasNext())
      list.add(Node.wrap((org.jdom.Content)iter.next()));
  }
  public int getLength() { return list.size(); }
  public Node item(int i) { return list.get(i); }
}
```

Fig. 8. Reimplementing DOM in terms of JDOM.

DOM's `getElementsByTagName` method *delegates* to JDOM's `getDescendants` method while creating a suitable JDOM element filter on the fly. The reimplementation of the `createElement` method does not refer to the wrapped document because the JDOM API uses regular constructor methods instead of document-centric factory methods. The reimplementation of the `getDocumentElement` method again delegates to the adaptee while it wraps the JDOM result before returning it. In general, "API-typed" arguments of the wrapper methods need to be unwrapped before being passed to the wrappee, and "API-typed" results returned by the wrappee methods need to be wrapped before being returned by the wrapper.

Consider the `NodeList` in Fig. 8. The class receives an `Iterator` upon construction which it immediately translates to a plain list (without wrapping the iterator until later). Hence, the iterator-based query is completed before even the first element is drawn from the node list. Consider the creation of a node list initiated by

```
public class Content {
  protected org.w3c.dom.Node domNode = null; // Adaptee
  protected String text = null; // State needed until DOM node construction

  public String getText() { return text; }
  public void setText(String text) { this.text = text; }

  // Needed by reimplementation
  Content() { }
  Content(String text) { this.text = text; }

  // Deferred construction of adaptee
  void build(Element parent) {
    domNode = parent.domNode.getOwnerDocument().createTextNode(text);
    ((org.w3c.dom.Element)parent.domNode).appendChild(domNode);
  }
}
public class Element extends Content {
  String name = null;
  private List<Content> kids = new LinkedList<Content>();

  public Element(String name) { this.name = name; }

  public Element addContent(Content elt) {
    kids.add(elt);
    if (domNode != null) elt.build(this);
    return this;
  }
  public String getName() {
    if (domNode != null)
      return ((org.w3c.dom.Element)domNode).getTagName();
    else
      return name;
  }
  public Content getChild(String name) { if ... }
  public void setText(String text) { if ... }
  public String getText() { if ... }

  // ... further build and wrapping infrastructure omitted ...
}
```

Fig. 9. Reimplementing JDOM in terms of DOM.

the `getElementsByTagName` method. Each of the returned JDOM descendants is wrapped with a new DOM identity.

3.2 Reimplementing JDOM in terms of DOM

Fig. 9 shows an attempt at reimplementing JDOM in terms of DOM. It is an attempt only because we cannot delegate node construction to DOM. More specifically, we cannot construct the wrappees right at the time of wrapper construction because DOM's node constructors rely on the document being part of the construction message. As a result, all subsequent messages sent to wrappers cannot be delegated to the wrappee either — unless the wrapper was parented in the meantime. Parenting is initiated by the `addContent` method, and "deferred" construction is described by the virtual `build` method of node types (shown for `Content` in the figure).

In an effort to still approximate a functional implementation of the JDOM interface, the code in the figure enriches the basic adapter pattern by extra state for the (yet) non-constructable adaptee: the `Content` class maintains the text value of the node, and the `Element` class maintains the element tag and the list of child elements. When a service is requested on an unparented node, then a designated implementation on that extra state is invoked instead of delegation to the adaptee. This approach is not very useful because we end up implementing the JDOM API from scratch.

3.3 Discussion

Our wrapper experiments make us conclude as follows:

Undelegatable services The wrapper approach cannot be applied when we encounter undelegatable services, as we did in the "JDOM in terms of DOM" case. In general, we assume that an undelegatable service is merely a consequence of different API *protocols* as opposed to different sets of API *services*. For instance, both DOM and JDOM support node construction, but their protocol slightly differs, and hence wrapping fails. The remaining discussion is limited to the cases where we succeed with a wrapper implementation, as we did in the "DOM in terms of JDOM" case (for the subset of services considered).

Wrapper proliferation If we were actually planning to use the wrapper implementation, we must be prepared to encounter behavioral deviations or efficiency issues due to the wrapper objects. In particular, (many) different wrapper identities may get associated with the same wrappee. Idioms that take a dependency on node identities will be broken — unless we engage in memoization of wrappers (as it is feasible for `getDocumentElement` of Fig. 8, for example) or an (expensive) global and bidirectional association map between wrappers and wrappees (as it is necessary for the constructor of the `NodeList` class of Fig. 8, for example).

Program specialization We consider it an important topic for future work to establish whether current program specializers (at the state of the art) [35, 5, 36] can be usefully adopted to convert applications (source code) on the grounds of the wrapper implementation. One challenging aspect of such an attempt is the requirement that the specializer must produce readable code that is comparable to the code that is written by a developer when performing a manual migration. We expect that the specializer needs to be configured for the API couple at hand. Also, the specializer needs to be aware of programming idioms that are relevant for the APIs at hand (such as different styles of consuming collections).

Semantic mismatch There is another fundamental problem with the wrapper approach: its premise to fully re-implement the source API, presumably up to observational equivalence. Our simplified wrapper implementations are faithful enough to pass the tests for the benchmark scenarios. However, independently developed APIs differ in semantic details that make it hard to achieve or to attest observational equivalence. (For instance, it is very hard to make XML serialization fully agree for any given couple of XML APIs.) When using straightforward delegation code, then the semantic differences will invalidate observational equivalence. When more complex delegation code is used to resolve the semantic differences, then inefficiency is incurred and program specialization is much less likely to be able to produce readable code. In the end, observational equivalence may still be limited to hold only for certain "usage scenarios" of the API, while the wrapper implementation has no way of quantifying these scenarios.

4 API migration by rewriting

We will now examine rewriting as a technique for API migration in the sense of performing a source-to-source translation. As we noted in the introduction, some sort of rewriting (or pattern-based replacement) has been used by a number of approaches that address some form of API migration [8, 24, 4, 30] without though covering the case where source and target APIs differ more substantially.

We are particularly interested in applying rewriting to the case where wrapping failed. The failure to delegate node construction for the "JDOM in terms of DOM" case actually suggests that local rewrite rules alone (as in say [4]) are insufficient to cover that case. Hence, we will consider rewrite rules that perform some sort of code reorganization and program analysis. The following specification focuses on the "Build XML document" scenario. We use the ASF+SDF approach to rewriting [6].[3]

4.1 Local rewrites

We begin with the part of the specification that models local rewrites. Here we assume that the rewrite function is parametrized in the document that is to be used for node construction. Given a sequence of API calls for XML processing, and (the variable that holds on) the hosting document, the statements are locally rewritten, one-by-one. Refer to Fig. 10. For instance, rewrite rule [setText] rewrites a call to JDOM's `setText` method to DOM's `setTextContent` method. The extra argument for the hosting document (see `&doc` in the figure) is used in rewrite rule [new-element] where JDOM's use of a regular constructor is replaced by the invocation of a factory method.

4.2 Code reorganization

For the local rewrite rules to be applicable, we need to determine (the variable that holds on) the hosting document. For simplicity, we assume here that the translation is

[3] ASF+SDF provides us with conditional term rewriting and support for concrete syntax (Java in our case). Conditional rewrite rules consist of any number of "premises" (conditions) and a conclusion separated by a horizontal line — that line is omitted when there are no conditions. Most rewrite rules of our specification are "unconditional". Conclusions are of this form: "*f(ArgTerm1, ..., ArgTermN) = Result*" — to be read from left to right: match terms on argument position and construct a result term. We only use premises of the following form: "*Variable := Term*" where the right-hand term is normalized and then bound to the left-hand side variable.

Types of the rewriting functions

```
xStms(BlockStm∗, doc:Id)  → BlockStm∗ // Rewrite many statements
xStm(BlockStm, doc:Id)  → BlockStm // Rewrite one statement
```

Rewrite rules

```
[new−element] xStm(org.jdom.Element &elt = new org.jdom.Element(&str);, &doc) =
                        org.w3c.dom.Element &elt = &doc.createElement(&str);

[addContent]  xStm(&elt1.addContent(&elt2);, &doc) =
                        &elt1.appendChild(&elt2);

[ setText ]  xStm(&elt.setText(&expr);,  &doc) =
                    &elt.setTextContent(&expr);

[ for−loop] xStm(for (&formal: &expr) { &stm∗ }, &doc) =
                      for (&formal: &expr) { xStms(&stm∗, &doc) }

[ default −xStm] xStm(&stm, &doc) = &stm

[xStms−nil] xStms(, &doc) =

[xStms−cons] xStms(&stm &stm∗, &doc) = xStm(&stm, &doc) xStms(&stm∗, &doc)
```

Fig. 10. Local rewrite rules for JDOM to DOM translation.

readily applied to a scope, in fact, a statement block, that lines up the entire "Build XML document" scenario — just like in Fig. 2. Hence, we need to locate the variable initialization for the relevant document in that statement block; c.f., *find-doc* in Fig. 11. That is, we search the statement block and return both the corresponding variable `&doc` and the *remaining* statements of the block.

If we compare the DOM and JDOM implementations of the "Build XML document" scenario, then we realize that the construction of the root element is done quite differently. The JDOM code *explicitly* constructs an element and *adds* it to the document. In contrast, the DOM code creates the root *implicitly* when it constructs the document, and *extracts* it for subsequent use. This difference is addressed by the following strategy. We use the *find-root* function (c.f., Fig. 11) to discover the variable for the root element — based on a call to the `addContent` method with the document as receiver. Further, we use the *find-tag* function (c.f., Fig. 11) to look up the construction statement for the root element; in fact, to look up the *tag* of the root element — based on the variable for the root that we obtained before. Both helpers remove the relevant statements from the block.

The rewrite rule in Fig. 12 puts together all the pieces. The incoming statement block is first transformed to remove the code for document and root construction. Then, the local rewrite rules are applied. Finally, a complete statement block is returned that begins with the construction of the document and the extraction of the root element (in DOM style).

Types of the rewriting functions

```
find−doc(BlockStm∗) →<BlockStm∗, Id>
find−root(BlockStm∗, Id) →<BlockStm∗, Id>
find−tag(BlockStm∗, Id) →<BlockStm∗, StringLiteral>
```

Find the statement that constructs and assigns the document.

```
[ find−doc] find−doc(org.jdom.Document &doc = new org.jdom.Document(); &stm∗)
              = <&stm∗, &doc>

[ default−find−doc] <&stm∗', &doc> := find−doc(&stm∗)
                    ─────────────────────────────────────────────
                    find−doc(&stm &stm∗) = <&stm &stm∗', &doc>
```

Find the statement that adds the root element to the document.

```
[ find−root] find−root(&doc.addContent(&root); &stm∗, &doc)
               = <&stm∗, &root>

[ default−find−root] <&stm∗', &root> := find−root(&stm∗, &doc)
                     ─────────────────────────────────────────────
                     find−root(&stm &stm∗, &doc) = <&stm &stm∗', &root>
```

Find the element construction for the root, and hence the root tag.

```
[ find−tag] find−tag(org.jdom.Element &root = new org.jdom.Element(&str); &stm∗, &root)
              = <&stm∗, &str>

[ default−find−tag] <&stm∗', &str> := find−tag(&stm∗, &root)
                    ─────────────────────────────────────────────
                    find−tag(&stm &stm∗, &root) = <&stm &stm∗', &str>
```

Fig. 11. Code-reorganization helpers for JDOM to DOM translation.

4.3 Discussion

The JDOM to DOM translation, to the extent shown, is prohibitively naive. The following discussion lists limitations in a systematic manner. Thereby, it gives an indication of the complexity of a proper solution.

Intra-procedural rewriting The transformation is intra-procedural, as it stands. For instance, we cannot apply it to the entire `makeDocument` method of Fig. 2; we are limited to reason about the translation of the method *body*. An inter-procedural generalization is feasible — with the usual caveat about dynamic dispatch, callbacks, and other difficult program constructs. That is, any method call, within the given scope of the translation, requires recursion into the body of the called method. Also, method signatures have to be converted by applying a type mapping like the following:

JDOM type	**DOM type**
org.jdom.Document	org.w3c.dom.Document
org.jdom.Element	org.w3c.dom.Element
...	...

```
Type of the rewriting function

jdom2dom(BlockStm*) → BlockStm*
```

```
Top-level rewrite rule of JDOM to DOM translation

[main] <&stm*1, &doc> := find−doc(&stm*),
        <&stm*2, &root> := find−root(&stm*1, &doc),
        <&stm*3, &str> := find−tag(&stm*2, &root)
        ------------------------------------------------------------
        jdom2dom(&stm*) =
            org.w3c.dom.Document &doc = DocumentBuilderFactory
                .newInstance()
                .newDocumentBuilder()
                .getDOMImplementation()
                .createDocument(null, &str, null);
            org.w3c.dom.Element &root = &doc.getDocumentElement();
            xStms(&stm*3, &doc)
```

Fig. 12. Main module of JDOM to DOM translation.

(This type mapping is already used implicitly by the local rewrite rules; see rewrite rule [new-element] in Fig. 10.) Clearly, if a method is reached by inter-procedural rewriting, and hence, if its signature is modified, then, obviously, *all callers* must be migrated; see the issue of a global strategy below.

Local, single, unaliased assignments The applicability of the present transformation relies on an escape analysis: only local variables must be used for holding onto XML objects. The transformation would be considerably more complicated when variables are allowed to escape from the relevant scope (e.g., when fields are used). Further, the correctness of the transformation relies on the fact that the variables for document and root are assigned to only once (also excluding aliasing).

Idiomatic incompleteness There are various idiomatic variations that are not covered by the simple transformation. One variation is method chaining as admitted by the JDOM API. For instance, child elements can be added within chains of calls to `addContent` method. Also, "substitution" (method calls with non-variable argument expressions) is an obvious variation; consider, for example, the construction of a node directly on the argument position of the `addContent` method. Further, other Java idioms come to mind, e.g., loops other than *for* loops. All these limitations are relatively easy to lift by either applying a code normalization prior to rewriting or by adding designated rewrite rules. (In the former case, normalization must be inverted after the actual translation because the code should be generally preserved as much as possible.)

API incompleteness The transformation only covers very few API methods and types. For simplicity, the default rule [default-xStm] of Fig. 10 simply skips all statements that do not match any of the patterns described by the other rules. That is:

```
[ default −xStm ]   xStm(&stm, &doc) = &stm
```

It is straightforward to replace this liberal default by a policy that avoids skipping over any statement that involves the source API. It is also straightforward to add local rewrite rules to cover a larger API subset (as long as no complicated, non-modular code reorganization issues arise).

Lack of a global translation strategy The transformation must be readily applied to a scope that is assumed to comply with the scenario "Build XML document". Two dimensions of generalization are needed to obtain a *global translation strategy* — referring here to the notion of strategy à la the language and system Stratego [41] for software transformation. First, the translation must be expanded to apply to an arbitrary source unit that may contain any number of scopes of interest. This dimension requires *iteration* over the scopes of the source unit; it also requires the incorporation of *applicability conditions* to decide whether the translation should be attempted at any given scope. Second, the translation must cover all scenarios of interest.

Unawareness of data dependencies Except for the ad-hoc treatment of document and root construction, the transformation does not express or check any data dependencies between the statements that are rewritten. For instance, there is the implicit assumption that the results from all calls of the element constructor, in the given scope, will be ultimately added to the same hosting document (at some level of nesting). When this is not the case, the translation will be incorrect indeed because it will invoke the factory method for element construction on a specific document (that was looked up by *find-doc*). We seek a way to express data dependencies between the API calls that presumably make up for some scenario; see the next section.

Lack of well-typedness preservation In the general case, it is unrealistic to expect a proof of semantics preservation for API migration specifications. However, we may want to establish other guarantees about a (rewriting-based) migration specification. Specifically, well-typedness preservation is desirable: when the migration of a (well-typed) program completes, then the result is guaranteed to be well-typed (without applying a separate phase of type checking). It may be possible to adopt ideas from the rewriting-based approach of Balaban et al. [4] where type constraints (as known from type checking [31]) are generated (from the input program and the rewrite rules) to determine where updates can be applied such that well-typedness is preserved. Related work on type-safe code generation [18, 22, 15] may be useful in this context, too. Here the observation is that some parts of a migration specification will essentially interpret a service of the source API as a kind of macro that expands to services of the target API.

5 API migration based on API protocols

The "amount of issues" with the (naive) rewriting approach suggests that we should generally seek extra "discipline" so that migration based on some form of rewriting becomes more manageable. In this section, we will discuss one disciplining measure: the use of *grammar-based API protocols*. The overall idea can be summarized as follows:

– We view API-usage scenarios as *languages over the terminals of execution traces*, i.e., actions for object construction or method invocation. The languages are described by context-free grammars (and eventually attribute grammars).
– Now we could use standard parsing techniques to decide whether a given program *run* meets a protocol. However, we are rather interested in the *static* property of a program to encode a scenario that meets the protocol. To this end, we re-interpret the grammars for the API protocols as descriptions of sets of *control-flow graphs* (CFGs), and we lift parsing from the trace level to the CFG level.
– At this point, we can *annotate protocols of the source API with actions of the target API* so that the enriched protocol defines a translation from CFGs over the source API to CFGs over the target API. We argue that such a translation is more manageable than a freewheeling rewriting-based specification.

5.1 Context-free approximation of API protocols

As a first approximation, consider the following EBNF-style grammar that describes the API-usage scenario underlying the benchmark scenario "Build XML document":

$$\begin{aligned} build &= \underline{\mathsf{newDocument}}\ \underline{\mathsf{newElement}}\ \underline{\mathsf{addContent}}\ content \\ content &= \underline{\mathsf{setText}}\ \mid\ (\underline{\mathsf{newElement}}\ content\ \underline{\mathsf{addContent}})* \end{aligned}$$

Here, we use (underlined) terminals to refer to the corresponding API method calls. That is, the API-usage scenario of building documents entails the construction of a new document, followed by the construction of an element (presumably the root element), followed by an action to add the root element to the document, followed by actions to build the content of the root element. We mention in passing that sequential composition can be complemented by permutational or interleaving composition — thereby enabling a more flexible order of actions.

5.2 API protocols

Context-free grammars are insufficient to capture data dependencies between the actions participating in a scenario. (This limitation corresponds to the *unawareness of data dependencies* in § 4.3.) Consider again the first few actions of *build*:

$$\underline{\mathsf{newDocument}}\ \underline{\mathsf{newElement}}\ \underline{\mathsf{addContent}}$$

This formulation does not ensure that the object constructed by $\underline{\mathsf{newElement}}$ is added to the object constructed by $\underline{\mathsf{newDocument}}$ via the method call $\underline{\mathsf{addContent}}$. As a remedy, we take into account the object identifiers for all arguments and results of all constructor and method calls. We generalize from context-free grammars to *attribute grammars* [27] so that we can constrain object identifiers in an API protocol. Fig. 13 shows the API protocol for the scenario of building XML documents. Based on the grammatical view on logic programming [13], we use Definite Clause Grammars (DCGs; [32, 38]) as the attribute grammar "notation" of choice. Thereby, the API protocols become directly executable in Prolog. To preserve EBNF style, we rely on higher-order predicates for EBNF's regular expression operators [28]; c.f., the use of the postfix (higher-order) predicate */1 for EBNF's iteration.

```
build(Doc) ⟶
    newDocument(Doc),            % Construct a new document with identity Doc.
    newElement(Root),            % Construct a root element with identity Root.
    addContent(Doc, Root),       % Add the root element to the document indeed.
    content(Root).               % Fill in the content of the root element.

content(Element) ⟶
    setText(Element).            % Set the text of a leaf element.

content(Parent) ⟶
    (child(Parent))*.            % Add many children.

child(Parent) ⟶
    newElement(Child),           % Construct a child element.
    content(Child),              % Fill in the content of the child element.
    addContent(Parent, Child).   % Add the child element to the parent at hand.
```

Fig. 13. The API protocol for building JDOM trees.

5.3 Basic actions

Even though we do not plan to ever *collect* execution traces, we will still define the possible actions that could occur in these traces so that we can reason about those actions at the level of control-flow graphs:

- $new(Oid, Type)$ (Object construction)
- $special(Result, Signature, Arguments)$ (Object initialization)
- $virtual(Result, Signature, Arguments)$ (Virtual method invocation)
- $static(Result, Signature, Arguments)$ (Static method invocation)

Oid is a placeholder for a constructed object identifier; $Type$ is a placeholder for a method signature; $Result$ and $Arguments$ are placeholders for object identifiers as well. Fig. 14 gives the part of the DCG that maps the apparent terminals of the API protocol for building JDOM trees (i.e., $newDocument$, $newElement$, $addContent$, and $setText$) to actions for method and constructor calls.[4]

5.4 Control-flow trees

Eventually, we want to determine whether a program encodes a a scenario that meets a given API protocol. To this end, we view programs as suitable control-flow graphs (CFGs) for which we need to check whether they are contained in the set of CFGs generated by the grammar for the API protocol.

We limit ourselves to a simple inter-procedural analysis for obtaining the CFGs of interest. That is, non-API calls must be non-virtual and non-recursive so that we can

[4] In DCG notation, right-hand side literals of the form *[T]* denote *terminals*. In the usual DCG example, a terminal is a character, a keyword, or a token parametrized by a token attribute. In our application, terminals are terms that represent actions as listed above.

```
newDocument(Doc) ⟶
    [new(Doc, 'org.jdom.Document')],
    [ special ( _,  '<org.jdom.Document: void <init>()>', [Doc])].

newElement(Element) ⟶
    [new(Element,  'org.jdom.Element ')],
    [ special ( _,  '<org.jdom.Element:  void  <init>(java.lang. String )>',[ Elt | _ ])].

addContent(Doc, Content)  ⟶
    [ virtual (Doc,
               '<org.jdom.Document: org.jdom.Document addContent(org.jdom.Content)>',
               [Doc,Content ])].

addContent(Parent,  Child)  ⟶
    [ virtual ( Parent,
               '<org.jdom.Element:  org.jdom.Element addContent(org.jdom.Content)>',
               [Element,  Child ])].

 setText (Element)  ⟶
    [ virtual (Element,
               '<org.jdom.Element:  org.jdom.Element  setText (java . lang. String )>',
               [Element,  _ ])).
```

Fig. 14. Basic actions of the API protocol for building JDOM trees.

inline the called methods into a simple intra-procedural CFG that is essentially of tree shape (hence, a "control-flow tree", i.e., a CF tree). The (slightly simplified) grammar of such CF trees is the following:

$$\begin{array}{rll} c = & \underline{\textsf{skip}} & \text{(The empty statement)} \\ \mid & a & \text{(Method and constructor calls)} \\ \mid & c\ \underline{\textsf{then}}\ c & \text{(Sequential composition)} \\ \mid & c\ \underline{\textsf{or}}\ c & \text{(Selective composition, say if ... else ...)} \\ \mid & \underline{\textsf{iterate}}\ c & \text{(Iteration, say loops)} \end{array}$$

This view on the program does not contain any "variable declarations". Instead, the actions a refer to *symbolic* object identifiers — subject to a symbolic execution [25] of the program. Here we leverage *enhanced* API signatures for the benefit of establishing constraints on arguments and results of API calls. For the scenario of building JDOM trees, we need to observe equality constraints like the following:

The result of `org.jdom.Element.addContent` equals the receiver.

Such equality constraints express JDOM's capability for method chaining. If we do not pay attention to these equality constraints, the CF trees contain extra symbolic object identifiers thereby interfering with the expected precision of expressing data dependencies in an API protocol. In other scenarios, we have also encountered the need for ownership constraints [9].

5.5 CF-tree parsing

The above grammar of CF trees resembles the regular operators that are used in forming the EBNF part of our API protocols. Such resemblance is the foundation of a strategy for parsing such CF trees in accordance with an API protocol. We start from a configuration $\langle c, v\rangle$ with c as the complete CF tree and v as the start symbol of the API protocol. In general, the v component is a generalized sentential form (i.e., a regular expression over terminals and nonterminals). We use a top-down parsing approach where leading nonterminals are immediately unfolded. (Hence, we do not admit left-recursion.) Parsing commences as follows:

- $\underline{\textsf{skip}}$ matches with ϵ.
- a matches with a.
- c $\underline{\textsf{then}}$ c' matches with $v\,v'$ if c matches with v, and c' matches with v'.
- c $\underline{\textsf{or}}$ c' matches with $v \mid v'$, if c matches with v, and c' matches with v'.
- $\underline{\textsf{iterate}}$ c matches with $v*$ if c matches with v.

We need an additional rule to deal with actions that are not of interest for the API protocol at hand. Here we assume that an API protocol is meant to cover all actions of types for one or several APIs, whereas the API protocol is meant to be oblivious to actions of all other types. The additional rule is to simply treat such uninteresting actions as ϵ.

We also need additional rules to cater for control flows that essentially correspond to "loop unrolling". That is, we cannot insist on the sentential form $v*$ to be matched only with the control flow $\underline{\textsf{iterate}}$ c. Likewise, we should admit control flows to select one alternative when the grammar provides several. Further, we need to define matching for optionality in the grammar ("?"). Hence, we add the following rules:

- c matches with $v \mid v'$ if c matches with v or with v'.
- c matches with $v?$ if c matches with $v \mid \epsilon$.
- c matches with $v*$ if c matches with $(v\,v*)?$.

These rules can be directly used for parsing CF trees with the DCGs for API protocols. The simple case of parsing lists for normal DCGs coincides with (say right-associative) sequential composition of actions.

5.6 Protocol-based translation

CF-tree parsing establishes whether a program encodes a scenario that meets a given API protocol. By enriching an API protocol, and by generalizing parsing to rewriting, we derive an effective translation for API migration.

We start from the protocol for a given scenario for the source API; see *at the top* of Fig. 15. We continue by making the protocol (the DCG) capture the parsed actions; see the intermediate step *in the middle of* of Fig. 15. The meta-predicate $>>$/2 captures the prefix consumed by its first argument as its second argument. All these captures are then composed as a synthesized attribute of the left-hand side. (In accordance with the general format admitted by CF trees, we do not need to flatten these lists.)

A rule of the API protocol for building JDOM trees; see Fig. 13.

```
jdom:build(Doc) ⟶
      jdom:newDocument(Doc),          % Construct a new document with identity Doc.
      jdom:newElement(Root),          % Construct a root element with identity Root.
      jdom:addContent(Doc, Root),     % Add the root element to the document indeed.
      jdom:content(Root).             % Fill in the content of the root element.
```

An illustrative, intermediate step

```
jdom:build(Doc,
      % Concatenation of reproduced matched actions
      [JDomNewDoc,JDomNewRoot,JDomAddContentToDoc,Content]
   ) ⟶
      jdom:newDocument(Doc)        >> JDomNewDoc,
      jdom:newElement(Root)        >> JDomNewRoot,
      jdom:addContent(Doc, Root)   >> JDomAddContentToDoc,
      jdom:content(Root, Doc, Content).
```

The corresponding part of the protocol-based translation

```
jdom:build(Doc,
   [ (JDomNewDoc, DomCreateDoc),
     (JDomNewRoot, DomGetRoot),
     (JDomAddContentToDoc, []),
     Content ]
   ) ⟶
            jdom:newDocument(Doc)        >> JDomNewDoc,
            jdom:newElement(Root)        >> JDomNewRoot,
            jdom:addContent(Doc, Root)   >> JDomAddContentToDoc,
            jdom:content(Root, Doc, Content),
            {
              dom:createDocument(Doc, DomCreateDoc),
              dom:getRoot(Doc, Root, DomGetRoot)
            }.
```

Helpers to construct associated DOM actions

```
dom:createDocument(Doc, [
    static (Factory, '<javax.xml.parsers.DocumentBuilderFactory: ...>', []),
    interface (Builder, '<javax.xml.parsers.DocumentBuilderFactory: ...>', [Factory]),
    interface (DomImpl, '<javax.xml.parsers.DocumentBuilder: ...>', [Builder]),
    virtual (Doc, '<org.w3c.dom.DOMImplementation: ...>', [DomImpl, _, _, _]) ]).

dom:getRoot(Doc, Root, [
    virtual (Root,
            '<org.w3c.dom.Document: org.w3c.dom.Element getDocumentElement()>',
            [Doc])]).
```

Fig. 15. A protocol-based translation from JDOM to DOM (excerpt).

Eventually, we *couple* the matched actions of the source API with the corresponding actions of the target API. Thereby, we describe the replacement of source actions by target actions including the precise location of new target actions; see the translation specification *at the bottom* of Fig. 15. The enhanced DCG synthesizes "coupled CF trees". We sense the potential for a domain-specific language to better express these translations.

5.7 Related work on API-usage scenarios and protocols

There is no generally agreed notion of API protocol. However, some options can be inferred from the various approaches that focus on *recovery* of some sort of API-usage scenarios. For instance, [23] recovers scenarios as UML sequence diagrams; [1] recovers scenarios as control-flow-sensitive, static, inter-procedural traces that are represented as partial orders; [37] recovers temporal API specifications of the form of finite-state automata by means of static analysis (abstract interpretation).

The description of API-usage scenarios is also relevant for guiding *code-search engines* [34, 21, 40, 42] or *code completion* [29]. The description and verification of the *correct use* of an API (or components in the sense of component-based software) [44, 12, 19] has suggested several kinds of constraints on API usage, e.g., sequencing, interleaving, ownership.

None of the above approaches relates to API *migration*, and it is not obvious how these sorts of protocols would need to be adopted to be useful in this context. The contribution of our proposal is that API-usage scenarios are described by grammars over execution traces with an alternative semantics that can be used to parse CF trees and further refined to rewrite these trees for the purpose of API migration.

In terms of modeling API protocols, our work is closely related to and inspired by Czarnecki, Antkiewicz et al.'s notion of framework-specific modeling languages (FSMLs; [3, 2]). FSMLs model usage scenarios of OO frameworks such as Java applets or Struts (but potentially also APIs). Linguistically, framework-specific metamodels are feature models (i.e., some form of grammars) over *abstract code structures* that are enriched by attribute constraints, and mapping information to facilitate forward engineering (i.e., code generation) and reverse engineering (i.e., extraction of feature configurations), while both directions are informed by additional program analyses. It has been shown that FSMLs are also suitable to support framework migration [7]. Compared to FSMLs, our contribution is a form of protocol whose interpretation is a set of control-flow graphs while the protocol can be optionally annotated to facilitate rewriting on control-flow graphs.

6 Conclusion

We have described a benchmark for API migration within the XML programming domain. The challenge is to be able to migrate programs that use one XML API to use instead another API. We have studied basic options for API migration: wrapping and a basic rewriting approach. These options illustrated the hardness of the problem. Ultimately, we have proposed the use of protocols for capturing API-usage scenarios and disciplining the description of translations for API migration. The proposed form of protocol is grammar-based and relies on grammar-based and compiler techniques to implement migration, i.e.: parsing, rewriting, and flow analysis.

By putting online an extended benchmark and some of our implementational experiments, we hope to motivate others to join us in the investigation of the API-migration problem. Considerable research effort will be needed to arrive at a general form of API protocol that covers more arbitrary domains and APIs. A key challenge is also the amalgamation of the protocol language with a sufficiently general as much as effective and transparent framework for program analysis so that protocols (and translations upon them) can involve data-flow constraints, control-flow constraints, non-escaping constraints, and others. The verification of the relative completeness of a translation as well as up-front guarantees such as well-typedness preservation also has an impact on the design of the protocol language.

References

1. M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 25–34. ACM, 2007.
2. M. Antkiewicz. *Framework-Specific Modeling Languages*. PhD thesis, University of Waterloo, Electrical and Computer Engineering, 2008.
3. M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering*, pages 214–223. ACM, 2007.
4. I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 265–279. ACM, 2005.
5. S. Bhatia, C. Consel, A.-F. L. Meur, and C. Pu. Automatic Specialization of Protocol Stacks in Operating System Kernels. In *Proceedings of 29th Annual IEEE Conference on Local Computer Networks (LCN 2004)*, pages 152–159. IEEE Computer Society, 2004.
6. M. van den Brand, A. van Deursen, J. Heering, H. d. Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings, Compiler Construction (CC'01)*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.
7. A. P. Cheema. Struts2JSF - framework migration in J2EE using Framework-Specific Modeling Languages. Master's thesis, University of Waterloo, 2007.
8. K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359. IEEE Computer Society, 1996.
9. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM, 1998.
10. I. Şavga and M. Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *GPCE '07: Proceedings of the 6th international conference on Generative Programming and Component Engineering*, pages 175–184. ACM, 2007.

11. I. Şavga, M. Rudolf, S. Götz, and U. Aßmann. Practical refactoring-based framework upgrade. In *GPCE '08: Proceedings of the 7th international conference on Generative Programming and Component Engineering*, pages 171–180. ACM, 2008.
12. P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. Model checking software with well-defined apis: the socket case. In *FMICS '05: Proceedings of the 10th international workshop on Formal Methods for Industrial Critical Systems*, pages 17–26. ACM, 2005.
13. P. Deransart and J. Maluszynski. *A Grammatical View of Logic Programming*. MIT Press, 1993.
14. D. Dig, S. Negara, V. Mohindra, and R. Johnson. Reba: refactoring-aware binary adaptation of evolving libraries. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 441–450. ACM, 2008.
15. M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative Programming and Component Engineering*, pages 275–284. ACM, 2006.
16. M. Fowler. Crossing Refactoring's Rubicon, 2001. Available online `http://martinfowler.com/articles/refactoringRubicon.html`.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
18. S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. *SIGPLAN Notices*, 36(10):74–85, 2001.
19. M. Gopinathan and S. K. Rajamani. Enforcing object protocols by combining static and runtime analysis. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, pages 245–260. ACM, 2008.
20. J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283. ACM, 2005.
21. R. Holmes, R. J. Walker, , and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering (TOSEM)*, 32(12):952–970, 2006.
22. S. S. Huang, D. Zook, and Y. Smaragdakis. Statically Safe Program Generation with SafeGen. In *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Proceedings*, volume 3676 of *LNCS*, pages 309–326. Springer, 2005.
23. J. Jiang, J. Koskinen, A. Ruokonen, and T. Systä. Constructing Usage Scenarios for API Redocumentation. In *Proceeedings of 15th International Conference on Program Comprehension (ICPC 2007)*, pages 259–264. IEEE Computer Society, 2007.
24. R. Keller and U. Hölzle. Binary component adaptation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 307–329. Springer, 1998.
25. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
26. A. Klusener, R. Lämmel, and C. Verhoef. Architectural modifications to deployed software. *Science of Computer Programming*, 54(2-3):143–211, 2005.
27. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
28. R. Lämmel and G. Riedewald. Prological Language Processing. In *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, volume 44 of *ENTCS*. Elsevier Science, Apr. 2001.
29. D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 48–61. ACM, 2005.

30. 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.
31. J. Palsberg and M. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.
32. F. C. N. Pereira and D. H. D. Warren. Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3):231–278, 1980.
33. J. H. Perkins. Automatically generating refactorings to support api evolution. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 111–114. ACM, 2005.
34. N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 413–430. ACM, 2006.
35. U. P. Schultz, J. L. Lawall, and C. Consel. Specialization Patterns. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated Software Engineering*, page 197. IEEE Computer Society, 2000.
36. U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):452–499, 2003.
37. S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 174–184. ACM, 2007.
38. C. M. Sperberg-McQueen. A brief introduction to definite clause grammars and definite clause translation grammars, 2004. A working paper prepared for the W3C XML Schema Working Group. Available online at `http://www.w3.org/People/cmsmcq/2004/lgintro.html`.
39. K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering*, pages 377–380. ACM, 2007.
40. S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.
41. E. Visser, Z. el Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 13–26. ACM, 1998.
42. T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining Software Repositories*, pages 54–57. ACM, 2006.
43. Z. Xing and E. Stroulia. The JDEvAn tool suite in support of object-oriented evolutionary development. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, pages 951–952. ACM, 2008.
44. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):292–333, 1997.