Reusing Metadata Across Components, Applications, and Languages

Myoungkyu Song[a,*], Eli Tilevich[b]

[a]*Department of Electrical and Computer Engineering, University of Texas at Austin, TX*
[b]*Department of Computer Science, Virginia Tech, Blacksburg, VA*

Abstract

Among the well-known means to increase programmer productivity and decrease development effort is systematic software reuse. Although large scale reuse remains an elusive goal, programmers have been successfully reusing individual software artifacts, including components, libraries, and specifications. One software artifact that is not amenable to reuse is metadata, which has become an essential part of modern software development. Specifically, mainstream metadata formats, including XML and Java 5 annotations, are not amenable to systematic reuse. As a result, software that uses metadata cannot fully reap the benefits traditionally associated with systematic reuse. To address this lack of metadata reusability, this article presents Pattern-Based Structural Expressions (PBSE), a new metadata format that is not only reusable, but also offers conciseness and maintainability advantages. PBSE can be reused both across individual program components and across entire applications. In addition, PBSE makes it possible to reuse metadata-expressed functionality across languages. In particular, we show how implementations of non-functional concerns (commonly expressed through metadata) in existing languages can be reused in emerging languages via automated metadata translation. Because metadata constitutes an intrinsic part of modern software applications, the ability to systematically reuse metadata is essential for improving programmer productivity and decreasing development effort.

Keywords:
Software reuse, metadata, non-functional concerns, annotations, XML, domain-specific language

1. Introduction

Whenever striving to improve software quality or increase productivity, programmers commonly resort to reusing existing software, a practice known as software reuse [1]. This practice refers to extracting reusable pieces from an existing software product to be integrated in a new software product. Reusing software rather than developing it from scratch increases programmer productivity [2]. Reusing tested software pieces in a software product improves the product's overall quality [3]. Applications in the same domain, such as banking, retail, government, and defense, commonly share common functionalities. The larger the domain, the more applications can potentially reuse these functionalities, thereby saving development effort and costs.

Because of its proven quality and productivity benefits, software reuse has received considerable attention from software researchers and practitioners alike. The research literature contains numerous examples of reusing a variety of software artifacts, including components [4, 5, 6, 7], libraries [8, 9, 10], and specifications [11, 12, 13]. In this article, we explore the issues of reusing yet another artifact of modern software construction—*framework metadata*, which comes in many formats, including XML, Java 5 annotations, C# attributes, and C/C++ pragmas.

Metadata has become an important building block of modern software applications. The prominent role of metadata is due to the software development process becoming increasingly declarative, particularly when it comes to implementing non-functional concerns (NFCs). Despite enabling the declarative programming model, existing metadata formats do not lend themselves easily to systematic reuse. In fact, major framework metadata formats—Java 5 annotations, XML configuration files, C# attributes—are crafted individually for different program components.

*Corresponding author
Email addresses: mksong1117@utexas.edu (Myoungkyu Song), tilevich@cs.vt.edu (Eli Tilevich)

For example, in JEE [14], XML deployment descriptors typically codify how individual Java classes interface with frameworks that implement various NFCs. In other words, programmers write XML files for each program class that uses any framework functionality. When using annotations, programmers must not only annotate individual programs separately, but they also have to individually annotate each class in the same program. This lack of metadata reusability constrains the power of the declarative model. Although declarations are shorter than procedural instructions, the necessity to repeat declarations not only creates unnecessary work for the programmer, but also increases the probability of introducing errors.

In this article, we first present the results of our analysis of why existing metadata formats are not easily reusable. Based on these results, we then present our new metadata format that is not only reusable, but also offers conciseness and comprehensibility advantages. Our new metadata format even enables cross language reuse of NFCs, a property that can benefit those emerging languages that are compiled to mainstream languages.

To improve metadata reusability, this article contributes the following novel insights:

- A clear exposition of the advantages and shortcomings of mainstream framework metadata formats—XML and language-integrated metadata.
- Pattern-Based Structural Expressions (PBSE)—a new metadata format that offers usability, reuse, and ease-of-evolution advantages, as compared to both XML and annotations.
- An automated translation approach that, given pattern-based structural expressions and their corresponding source files, can annotate the source with equivalent Java 5 annotations.
- An approach to reusing non-functional concern implementations of a mainstream language from an emerging language program, when the emerging language is compiled to the mainstream language;
- Automated cross-language metadata translation—a novel approach to translating metadata alongside compiling the source language;
- *Meta-metadata*, a domain-specific language that declaratively expresses how one metadata format can be translated into another metadata format;
- An approach to unit test and transparently persist X10[1] programs for both Java and C++ backends, the X10 compilation targets.

The remainder of this article is structured as follows. Section 2 gives the background and motivates this work. Section 3 presents Pattern-Based Structural Expressions (PBSE), our new metadata format. Section 4 demonstrates the reusability benefits of PBSE in the context of source-to-source compilation. Section 5 evaluates this work through case studies. Section 6 discusses the advantages and shortcomings of this work. Section 7 compares this work to the existing state of the art. Section 8 presents concluding remarks.

2. Background and Motivation

This article is concerned with systematic metadata reuse. In modern software development, metadata is primarily used to expresses non-functional concerns. These concerns are commonly implemented by means of software frameworks. In our work, we develop a new domain-specific language to express metadata. Next, we first introduce these technologies, and then present a motivating example that demonstrates the utility of our approach.

2.1. Background

A software application consists of a set of functionalities commonly referred to as *concerns*. Based on the role that concerns play in an application, they are traditionally labeled as functional or non-functional. Functional concerns implement the application's business logic; they define *what* the application does. Non-functional concerns (NFCs) qualify functional concerns; they define *how* the application executes its business logic. For example, the same business logic can be executed with different levels of persistence, security, transactions, or fault-tolerance—each implemented as a separate NFC and playing an essential and critical role in modern software applications.

[1] X10 is an emerging language being developed at IBM Research. The X10 compiler compiles an X10 program to both Java and C++.

To deliver quality software under tight deadlines, programmers leverage object-oriented frameworks to implement NFCs. These frameworks are steadily transitioning toward *the declarative programming model*, in which programmers express NFCs using metadata, while frameworks implement them. Framework metadata—a medium for expressing how program constructs interact with a framework—comes in different formats. One such format is XML, used for writing stand-alone XML configuration files, sometimes called deployment descriptors in JEE. More recently, metadata tags have been introduced into mainstream programming languages, including Java and C#. Being placed near program constructs, Java annotations are a part of the source. Although in their latest releases, some frameworks have been changing their metadata format from XML to language-integrated metadata formats, both formats remain in wide use in modern software development.

Domain-specific languages (DSLs) reduce the complexity of constructing software in a particular domain by providing domain-specific syntactic abstractions. In other words, a DSL is tailored toward the problems in a specific domain, such as database programming—SQL, statistical computations—R, and text typesetting—LaTeX. As compared to general purpose languages, a DSL can solve a domain problem more concisely with a reduced programming effort [15, 16]. DSLs have been successfully employed to solve software engineering challenges in multiple domains. Our approach leverages DSLs to create a metadata format that can be systematically reused across components, applications, and languages.

2.2. Shortcomings of Existing Metadata

As we argue next, both mainstream language-integrated and external metadata formats have limitations that hinder both the development and evolution stages of the software engineering process. Particularly harmful is the limitation that metadata cannot be systematically reused.

As an example of using metadata in a modern software application, consider the two code snippets using transparent persistence that appear in Figure 1. On the left, class `ManagerBean` is persisted using EJB 2 [17], a framework that uses XML configuration files to specify how instances of Enterprise Java Beans should be mapped to relational database tables. On the right, class `ManagerEJB`, with equivalent functionality, is persisted using EJB 3 [18], another persistence framework that uses annotations.

```
class ManagerBean {
  public String getOrderId () {..}
  public String getStatus () {..}
  public void setOrderId (String p) {
  ..}
  public void setStatus (String p) {
  ..}
}
```

```
<entity>
  <ejb-class>ManagerBean</ejb-class>
  <abstract-schema-name>Manager
  </abstract-schema-name>
  <cmp-field><field-name>orderId
    </field-name></cmp-field>
  <cmp-field><field-name>status
    </field-name></cmp-field>
  <primkey-field>orderId
  </primkey-field>
  ...
</entity>
```

(1) Metadata as an XML file.

```
@Entity
@Table(name="Manager")
public class ManagerEJB {
  private String orderId;
  private String status;

  @Id
  @Column(name="orderId", primaryKey=true)
  public String getOrderId(){
    return orderId;
  }
  @Column(name="status", primaryKey=false)
  public String getStatus(){
    return status;
  }
  public void setOrderId(String parm){
    orderId = parm;
  }
  public void setStatus(String parm){
    status = parm;
  }
}
```

(2) Metadata annotations.

Figure 1: Transparent Persistence Framework Example.

2.2.1. Programmability

Using either metadata format to create correct specifications can be challenging. Authoring XML files with a text editor is cumbersome: the programmer must ensure not only the correctness of XML tags and grammar, but also the correspondence between the XML data and the program constructs of the persisted class. For example, each persisted class must be specified within `<entity> </entity>` tags, which is the root of an XML subtree with the descendant tags `<ejb-class>`, `<abstract-schema-name>`, `<field-name>`, `<primkey-field>`, etc. The immediate descendants of the `<entity>` tag may also have other descendants, making the authoring of such a tree-shaped structure quite error-prone. Although XML schemas can catch the deviations from the specified grammars, they cannot detect all the errors resulting from manual editing, leading to confusing compile and runtime errors. For example, forgetting to specify a `<primkey-field>` would render the entire specification invalid. Mistyping any field name would result in runtime errors, in response to the framework trying to access a non-existing field.

As part of the Java language, annotations are more straightforward than XML files to use. Nevertheless, adding annotations according to a convention set by a particular framework may quickly become difficult. Even though a code completion facility of a modern Integrated Development Environment (IDE) (e.g., Eclipse, IntelliJ, Netbeans, etc.) could help programmers enter well-formed annotations with correct attributes, code completion cannot help them determine the value of string attributes (e.g., name in the `@Column` annotation) or identify where annotations should be added. Programmers must ascertain these requirements based on implicit framework conventions. For example, the `@Column` annotation can be added either to persistent field or to their getter methods according to the JavaBean naming convention, and these two approaches cannot be mixed.

One could argue that more sophisticated IDE support and better programming tools in general could simplify the authoring of both XML files and annotations, but the very fact that such advanced support is required is a testament to the inherent complexity of expressing metadata using these formats.

2.2.2. Understandability

Consider a programmer assigned to take over an existing codebase written using a framework dependent on metadata. Examining XML files by hand is quite tedious. As a format, XML is optimized for automated computer processing rather than for exposing information intuitively to the programmer. To understand how a framework implements some functionality, XML files must be examined with their corresponding source files. For example, to understand how instances of class `ManagerBean` are persisted, both its source and XML deployment descriptor must be examined.

While annotations ease program understanding in the small, programmers must examine the entire framework-dependent codebase. Annotations scattered around the codebase, provide no structural or summary information. For example, annotations cannot explicitly express that all `private` fields of class `ManagerEJB` should be persisted. To determine this program fact, programmers must examine each getter method for the presence of the `@Column` annotation.

Sophisticated code analysis tools can certainly help programmers understand framework-dependent source code, but a more expressive metadata representation can render such analysis tools unnecessary.

2.2.3. Maintainability

Both metadata formats complicate maintenance. XML files are separate from the main source code, and their correspondences cannot be enforced by the compiler. If a source code change is not properly synchronized with the corresponding XML file, the problem will only be discovered at runtime. For example, if the `status` field in class `ManagerBean` is renamed to `orderStatus`, one must upgrade the corresponding entry in the XML file. This requirement, however, is implicit and depends entirely on the maintenance programmer.

Being a part of the language, annotations are easier to maintain. In the presence of structural changes, however, annotations must be added or removed accordingly. For example, when a field `time` is added both to the database and the `ManagerEJB` class, this field or its getter method must be properly annotated, lest it will not be persisted. Annotations cannot express that all fields sharing names with database columns are to be persisted.

As a result, maintaining framework-based applications is a formidable challenge that has been the target of several concerted research efforts [19, 20, 21]. These efforts, however, could be simplified if more expressive metadata could explicitly encode dependencies that must be preserved during program evolution.

2.2.4. Reusability

Reusability Across Components and Applications. Both XML and annotation-based metadata representations are not reusable. The persistence information must be explicitly encoded for each class. In Hibernate framework [22], another transparent persistence framework, a separate XML file must be used for each persistent class. In EJB 2, the same XML file is used for all the classes. Nevertheless, in both frameworks the persistence information is specified individually for each class. For example, the XML file for class `ManagerEJB` would not work with any other class. As a consequence, the programming effort expended on specifying how to persist `ManagerEJB` cannot be reused even for the classes in the same application.

Annotations do not improve reusability—each individual class must be annotated anew. The effort expended on annotating a class cannot be reused in annotating other classes, as annotations do not express any structural information. Each annotation expresses information only about the immediate program construct it annotates. Annotations cannot express the relationships between the annotated elements[2] [23, 24, 25] (e.g., all the fields annotated with the same annotation). Further, annotations cannot even express a naming equivalence between its attributes and the annotated program constructs. For example, the attribute `name` of annotation `Column`[3] directly corresponds to the name of the field of the getter method it is annotating. Because this correspondence is only implicit, it cannot be reused in other contexts, such as different classes or another application.

Reusability Across Languages. Although source-to-source language translation makes it possible to reuse code between source and target languages, the benefits of reuse do not extend to metadata. As an example, consider the code snippet on the left in Figure 2. It contains a skeletal representation of a test method written according to the conventions of the popular JUnit framework. This framework requires that test case methods be tagged with the annotation `@Test`. Source-to-source translation tools (e.g., Java Language Conversion Assistant (JLCA) [26]) can automatically translate Java to C#, thereby effectively reusing the programming effort expended on producing the original code. Unfortunately, the benefits of source-to-source translation do not extend to metadata. One way to translate this test case would be how it appears on the right of Figure 2. This code snippet shows the translated test case adhering to the conventions of the csUnit [27] C# unit testing framework. This framework requires that test case methods be tagged with the attribute `[TEST]`. Guided only by the syntactic correspondences between the source and target languages, source-to-source translators cannot map the JUnit annotation `@Test` to the C# attribute `[TEST]`. This mapping lies beyond any syntactic correspondences, as it stems from the domain-specific equivalence between the JUnit and csUnit unit testing frameworks. In fact, the JLCA tool does not even attempt to translate Java annotations to C# attributes. Even though the functionality specified through metadata may be equivalent in the original and source-to-source translated program versions, the programming effort expended on adding the metadata to the source language program cannot be reused in the automatically translated target language program.

```
1 @Test
2 public void someTestMethod() {
3 ...
4 }
```

```
1 [TEST]
2 public void someTestMethod() {
3 ...
4 }
```

Figure 2: Translating a test case from Java JUnit (left) to C# csUnit (right).

All in all, existing metadata formats unnecessarily complicate program development and evolution. The biggest shortcoming of these metadata formats is that metadata is not reusable across components, applications, or languages. We posit that a metadata format designed with reusability in mind can also provide other software engineering benefits.

[2] Although some annotations (e.g., The `@Inherited` and `@MappedSuperclass`) express the inheritance relationship between two concrete classes, and as such cannot be reused.

[3] When the `Column` annotation is omitted, the framework, by default, uses the same names for class fields and table columns. However, the differences in naming conventions between relational databases and object-oriented languages make this default option rarely applicable.

3. Pattern-Based Structural Expressions

Next we explain the design space we have considered to create a new metadata format that we call Pattern-Based Structural Expressions (PBSE). Then we explain how PBSE enables effective metadata reuse across components, applications, and languages.

3.1. Design

By examining metadata specifications of frameworks from different domains across multiple applications, we have observed that metadata is added to a program following well-defined patterns. It is this observation that led us to a new metadata format that is not only more concise and expressive, but also reusable. Our new metadata format, called Pattern-Based Structural Expressions (PBSE), is introduced by example next.

```
1  public class ManagerEJB {
2    private String orderId;
3    private String status;
4
5    public String getOrderId(){
6      return orderId;
7    }
8    public String getStatus(){
9      return status;
10   }
11   public void setOrderId(String parm){
12     orderId = parm;
13   }
14   public void setStatus(String parm){
15     status = parm;
16   }
17 }
```

(1) Java code for PBSE in (2).

```
1  Metadata MyJPA<Package p>
2    Class c in p
3      Where (public *EJB)
4        c += @Table
5        @Table.name = (c.name =~ s/EJB$///)
6        Column<c>
7  Metadata Column<Class c>
8    Method m in c
9      Where (public * get* ())
10       m += @Column
11       @Column.name =
12         (m.name =~ s/^get///^[A-Z]///[a-z ]///)
13       Where (public * get*Id ())
14         @Column.primaryKey = true
15         m += @Id
16 MyJPA <"package1">
```

(2) PBSE metadata.

Figure 3: Transparent Persistence Framework PBSE Example.

Consider the original motivating example from the transparent persistence domain. An equivalent PBSE specification on Figure 3 (1) is applied to the Java code on the left. PBSE reuses Java 5 annotations, declared as special Java interfaces, to define its own metadata specifications. A key difference is that PBSE metadata is declared in standalone specification files that are kept separate from the Java source files.

PBSE specifications are organized into modules, each representing metadata for a particular program construct. A PBSE module starts with the keyword `Metadata` followed by the module's name and its parameter declaration. Figure 3 (2) contains two PBSE modules. The first module is called `MyJPA`, and it defines Java Persistence API metadata for a package `p`. Line 2 iterates over all the classes in the package. To designate that only the classes that are `public` and whose suffix is "`EJB`" are to be persisted, a `Where` clause is used with the pattern parameter `public *EJB`.

The indented `Where` clause specifies the metadata information that should be applied to the classes that match the clause's pattern. Line 4 attaches `@Table` metadata to the matched class. PBSE uses the `+=` operator to express the attaching of metadata to program constructs. Line 5 uses regular expressions—we use the Perl language regular expression style due to its wide adoption, in order to assign the value of the class name without its "`EJB`" suffix to the `name` property of the `@Table` metadata. The `=~` operator applies the regular expressions of its right operand to its left string operand. Line 6 invokes another PBSE module `Column` passing it the matched class as a parameter. All the invocations in PBSE are statically bound and are resolved by matching their names.

The `Column` PBSE module accepts a `Class` parameter and iterates over its methods (line 7). The `Where` clause on line 9 matches the getter methods following the JavaBean naming convention.[4] Line 10 attaches the `@Column` metadata to

[4] A more elaborate pattern could have filtered out the methods staring with "`get`" but returning `void`, e.g., `void getUpset()`.

the matched methods, and line 11 sets the `name` property of this metadata to the name of the method, having removed the "`get`" prefix and then changed the first letter to lowercase. Regular expressions are applied one after another from left to right, using the result of applying one expression as input to the next expression. Line 13 encodes the naming strategy for the getter methods which return persistent values, corresponding to the primary key of the underlying database table. The strategy in place assumes that the names of such getter methods will end with "`id`." The methods matched by this `Where` clause have the `primaryKey` property of their `@Column` metadata set to `true`, and another metadata item, `@Id`, is attached.

Finally, line 16 applies the `MyJPA` metadata module to package "`package1`." Thus, the persistence metadata will be attached to all the classes in this package. Further, since all persistent classes in an application usually share the same structure and naming conventions, `MyJPA` can be effortlessly reattached to other packages by adding only one line of code (e.g., `MyJPA<"package2">`).

3.2. Reusing PBSE Metadata Across Components and Applications

The applicability of PBSE is not confined to persistence frameworks. We have discovered that frameworks commonly use structural patterns in their metadata. Not all of these frameworks use both XML and annotations as their metadata formats. Therefore, in the following we compare our PBSE specifications to whichever metadata format is used by a given example framework.

3.2.1. JUnit

Unit testing frameworks exercise test methods designated as such using metadata. A well-known and widely-used unit testing framework is JUnit [28], which uses `@Test` annotations to designate test methods, and `@Before` and `@After` annotations to designate the setup and tear down methods for each test class. The `@Test` annotation, in particular, has to be added to each and every test method, which can be wasteful, particularly if the number of test methods is large. Furthermore, neglecting to annotate a newly-added test method will result in missing tests.

Figure 4 shows how JUnit metadata can be attached to all the test classes in a package, defined as all the public classes whose name starts with prefix "`Test`." The `@Test`, `@Before`, and `@After` metadata are attached to the public methods returning `void` in the test methods based on their respective prefixes. A more general pattern could match the test methods whose name does not start with the "`test`" prefix.

3.2.2. TestNG

TestNG [29] is another annotation-based unit testing framework, whose annotation set differs from that of JUnit. One could imagine how the necessity to change the annotations throughout an entire application from JUnit to TestNG or vice versa could preclude switching to another unit testing framework, even if such a switch is beneficial for

```
1  Metadata MyJUnitSuite <Package p>
2     Class c in p
3        Where (public class Test*)
4           MyJUnitTest <c>
5
6  Metadata MyJUnitTest <Class c>
7     Method m in c
8        Where (public void before* ())
9           m += @Before
10       Where (public void after* ())
11          m += @After
12       Where (public void test* ())
13          m += @Test
14
15 MyJUnitSuite <"package1">
```

Figure 4: PBSE metadata for JUnit framework.

```
1  Metadata MyTestNGSuite <Package p>
2     Class c in p
3        Where (public class Test*)
4           MyTestNG <c>
5
6  Metadata MyTestNG <Class c>
7     Method m in c
8        Where (public void before* ())
9           m += @BeforeMethod
10       Where (public void after* ())
11          m += @AfterMethod
12       Where (public void test* ())
13          m += @Test
14       Where (public void set* ( * ))
15          m += @Parameter
16          @Parameter.value = (m.name=~s/^set//)
17
18 MyTestingNG <"package1">
```

Figure 5: PBSE metadata for TestNG framework.

technical reasons. Not changing vendors solely due to prohibitive upgrade costs is described by the Vendor Lock-in anti-pattern [30].

PBSE, being external to the main source code, removes this anti-pattern. Figure 5 shows the PBSE metadata specification for TestNG applied to the same set of test classes as in the JUnit example above. As a more recent unit testing framework, TestNG offers additional capabilities, among which is the ability to set custom parameters for test classes. The additional rule starts on line 14, which attaches the metadata `@Parameter` to setter methods (line 15), and sets its `value` property to the field name designated by the getter method (line 16), according to the JavaBean naming convention.

Thus, the same application can be tested with JUnit or TestNG simply by using a different PBSE specification.

3.2.3. The Security Annotation Framework

Another framework domain that can benefit from our pattern-based approach to expressing metadata is security. The security functionality of a typical framework application is divided into access control and encryption. An example of a security framework for applications is the Security Annotation Framework (SAF)[31]. SAF provides access control and encryption functionality, both of which are configured using Java 5 annotations. Methods can be granted read, update, create, and delete access. When the code to be secured with SAF follows a naming convention, the access can be granted based on patterns over method names rather than for each individual method.

Figure 6 shows the PBSE metadata security specification for a package in which classes are written according to the Java Bean naming convention. In addition, these classes have factory methods, which start with the "`create`" prefix. Finally, methods with the "`delete`" prefix deallocate systems resources passed to them as parameters.

The access control policy expressed by this specification controls access for every public class by using the `@SecureObject` metadata. The `@Secure` metadata and its `SecureAction` property are attached as follows. Every getter method is given the `READ` access, while every setter method as well as any method starting with prefixes "`add`" and "`remove`" are given the `WRITE` access. The `DELETE` access is given to reference parameters of the methods whose name starts with the "`delete`" prefix. We borrow the AspectJ syntax of `Object+` to express reference types.

Enforcing a consistent access policy requires that the entire codebase be annotated thoroughly, without any tolerance for omissions or mistakes. For example, giving the `UPDATE` permission to a wrong method may breach security. Naming conventions have become a mainstay of industrial software development to the degree that they are often

```
1  Metadata MySecurity <Package p>
2     Class c in p
3        Where (public class *)
4           c += @SecureObject
5           MySecureObject <c>
6
7  Metadata MySecureObject <Class c>
8     Method m in c
9        Where (public * get* ())
10          m += @Secure
11          @Secure.SecureAction = READ
12       Where (public void [set|add|remove]*)
13          m += @Secure
14          @Secure.SecureAction = UPDATE
15       Where (static public Object+ create*)
16          m += @Secure
17          @Secure.SecureAction = CREATE
18       Where (public * delete* )
19          Parameter p in m
20          p += @Secure
21          Where (Object+ *)
22            p += (@Secure.SecureAction = DELETE)
23
24 MySecurity <"package1">
```

Figure 6: PBSE metadata for security framework.

```
1  Metadata MyWebService <Package p,
2                         Pattern classPattern>
3     Class c in p
4        Where (classPattern)
5           c += @WebService
6           @WebService.name =
7             (c.name =~ s/Impl$//)
8           Field f in c
9              Where (private * *)
10                f += @Autowired
11          Method m in c
12             Where (public * * ())
13                m += @WebMethod
14                @WebMethod.name = m.name
15
16 MyWebService <"package1",
17                "public class *Impl">
```

Figure 7: PBSE metadata for Spring Web Service.

enforced with automatic checkers. Integrated with a source control system, such an automatic checker can prevent committing any code edits that violate the naming convention in place. In light of that, applying a security policy based on structural patterns of the established naming convention is likely to prove more reliable than annotating methods individually.

3.2.4. Java Web Services

To support the ever-growing need for service-oriented applications, frameworks have been introduced to facilitate the exposition of regular classes as services. In particular, the Java Web Services (JWS) framework [32] provides a set of annotations that can be added to Java classes, methods, and fields, leaving it up to the underlying framework to provide the necessary plumbing to expose the annotated classes as externally-accessible Web services. If a class to be exposed as a Web service has many methods, each of them must be annotated individually.

Figure 7 shows the PBSE metadata specification that can be used to render all the public classes in a package as Web services. In particular, the logic required to annotate multiple classes and methods is expressed in only 12 lines of PBSE. The parameter pattern expressed by this specification encode that the name of a Web service will differ from that of its corresponding class by the "`impl`" suffix (line 17). Unlike other examples, the example in Figure 7 shows how the PBSE specification can take a pattern parameter. In this case, the parameterized name is that of the class name. Instead of hardcoding the name of Web service classes to a specific naming pattern, this PBSE specifications can be applied to classes adhering to any naming convention (e.g., having the "`Impl`" suffix or the "`Web`" prefix). This additional parameterization further increases the reusability of PBSE. The `@Autowired` metadata is attached to all the private fields. And public methods are expressed as corresponding to Web service methods with the same names.

3.3. Reusing PBSE Metadata Across Languages

A common implementation strategy for emerging programming languages is to compile them to some existing language. Source-to-source compilation is more straightforward than providing a dedicated compiler backend. Additionally, because mainstream, commercial programming languages have been highly optimized, compiling an emerging language to a mainstream one can produce efficient execution without an extensive optimization effort. The emerging languages that compile to mainstream languages or bytecode include Scala [33], JRuby [34], Jython [35], and X10 [36].

Because a source-to-source compiler can only directly translate a program from the source language to the target language, the NFC implementations in the target language cannot be accessed from the source language. Provided as libraries and frameworks in the target language, these implementations can be accessed only by declaring appropriate metadata for target language programs. As a result, emerging languages must reimplement all the NFCs from scratch.

We posit that it is possible to translate metadata alongside compiling the source language. Our approach requires not only expressive languages to specify metadata, but also how metadata is to be translated. To that end, we have extended PBSE to compile across languages, as specified by declarative translation strategies, to work with target language programs.

Our approach to reusing NFC implementations across languages works as follows. Rather than reimplement an NFC in an emerging language, the programmer can reuse the existing target language implementations. The approach enables the programmer to specify the needed NFC in a source language program by declaring metadata. The declared metadata is then automatically translated, so that the needed NFC implementation in the target language can be reused. If the source language compiles to multiple target languages, the NFC implementations can be reused for each target language.

The X10 Source-to-Source Compiler. The X10 compiler compiles X10 programs to Java and C++ backends. Because NFCs are expressed declaratively through metadata, a source-to-source compiler cannot emit code for their standardized implementations. Thus, to reuse NFC implementations, metadata translation must supplement source compilation.

To demonstrate the problem concretely, consider writing an X10 program. The X10 compiler translates X10 programs to either a C++ or Java backend. At some point, the programmer realizes that some portion of the program's state must be persisted. In other words, certain X10 object fields need to be mapped to the columns of a database table, managed by a Relational Database Management System (RDBMS). As the program is being developed, the persistent state may change with respect to both the included fields and their types. In terms of persistent storage, it is desirable

```
package model;

public class FmmModel {
  private var modelId:Long;
  private var energy:Double;
  // ...
  public def this(modelId : Long,
                  energy : Double, ..) {
    this.modelId = modelId;
    this.energy = energy;
    // ...
  }
  // ...
}
```

Figure 8: An X10 class to be compiled to Java and C++.

```
Metadata PersistentModelClasses<Package p>
  Class c in p
  Where (public class *Model)
    c += @table
    @table.name = c.name
    @table.class = c.name
    Field<c>
Metadata Field<Class c>
  Field f in c
  Where (private var *:*)
    f += @column
    @column.name = (f.name=~s/[a-z]/[A-Z]/)

PersistentModelClasses <"model">
```

Figure 9: Metadata to persist the X10 class in Figure 8.

for the C++ and Java backends to share the same RDBMS schema. This way, the state persisted by the Java backend can be used by the C++ backend and vice versa.

These requirements are quite common for modern software applications, and mainstream programming languages have well-defined solutions that satisfy these requirements. In particular, object-relational mapping (ORM) systems have been developed for all major languages, including Java and C++. Commercial ORM systems implement the NFC of transparent persistence. As we have seen above, an ORM system persists language objects to a relational database based on some declarative metadata specification, so that the programmer does not have to deal with tables, columns, and SQL. However, because X10 is an emerging language, an ORM system has not been developed for it. Developing an ORM system is a challenging undertaking for any language, but for X10 it would be even more complicated. Because X10 is compiled to Java or C++, an X10 ORM solution must be compatible with both of these compilation target languages.

The approach we present here can add transparent persistence to X10 programs by leveraging existing ORM solutions developed for Java and C++. To demonstrate how our approach works from the programmer's perspective, consider the X10 code snippet in Figure 8.

This figure depicts the X10 class `FmmModel` [37] that contains fields of different types. The number of fields and their types are likely to change as the program is maintained and evolved. Furthermore, our compilation target changes repeatedly between the Java and C++ backends. We need to persist the `private` fields of this class to a relational database according to the following naming convention. The class and the table share the same name, while the columns have the same names as the fields, but capitalized.

To that end, the programmer writes a PBSE metadata specification listed in Figure 9. The specification on the right expresses that all `private` fields of classes with suffix "Model" should be persisted. This PBSE specification constitutes all the manually written code that the programmer has to write to use our approach.

Based on a PBSE specification, our automated code generation tools produce all the necessary functionality to persist the fields of class `FmmModel` in an RDBMS for the Java and C++ backends. The automatically generated code artifacts include:

1. An X10 class called `TP` (short for **T**ransparent**P**ersistence) that provides an X10 API for saving and restoring the persistent fields. This class encapsulates all the low-level database interaction functionality such as transactions and can be further modified by expert programmers.
2. An XML deployment descriptor required by the Java Data Objects (JDO) ORM system [38]. The descriptor specifies how JDO should persist the fields of the Java class emitted by the X10 compiler for the Java backend.
3. A C++ header file that contains `#pragma` declarations required by ODB,[5] an open source ORM system for C++ [39]. The `pragma` declarations specify how ODB should persist the fields of the C++ class emitted by the X10 compiler for the C++ backend.

[5] Surprisingly, ODB is not an acronym.

When either the Java or C++ target of the X10 program executes, the fields in class `FmmModel` are transparently persisted to a database table. Our approach is highly customizable and configurable. Any Java or C++ ORM solution can be used by changing a configuration file. The code generated for the `TP` class can be easily modified by editing our code generation template. Finally, if the X10 compiler were to be extended for yet another cross language, our approach can be easily extended to transparently persist the target code, as long as the new target language has an ORM solution.

3.4. PBSE Language

Having seen all the advantages of PBSE over annotations and XML, one may wish that PBSE becomes a new de-facto metadata standard. Such a transition, however, would require multiple divergent stakeholders to come to a consensus. Thus, to make PBSE specifications immediately available to the framework programmer, we have implemented an automated translation tool that annotates Java source code based on its PBSE specification.

3.4.1. Language Summary

Figure 10 summarizes the syntax of PBSE. The language follows a minimalistic design, introducing new constructs only if necessary, with the goal of making it easier to learn and understand. For example, the class iterator can be used for iterating through both classes and interfaces of a package. PBSE expresses metadata declaratively and does not have explicit conditional or looping constructs. Nevertheless, a PBSE specification does contain a sufficient level of detail to describe the metadata information of a typical modern framework.

3.4.2. Translation Semantics

Next we treat the translation process from PBSE to annotations more formally. Figure 11 lists the symbols used in describing the translation process. The sets of program's structural constructs appear first. The structure of a program is defined by its classes, methods, method parameters, and fields, all of which are finite sets. Each of these structural

---

- **Metadata** *module_name*
  < [**Package**|**Class**|**Method**|**Field**|**Parameter**|**ProgramConstructPattern**]
  programConstructVariable >
  ...
  module_name<*program_construct_variable*>
  ...
  `PBSE can call another module passing a parameter.`

- [**Class**|**Method**|**Field**|**Parameter**] *iter_var* **in** *collection*
  `An iterator for a collection of program constructs.`

- **Where** ([*class_pattern*|*method_pattern*|*field_pattern*|*parameter_pattern*])
  `Patterns to match declarations of program constructs.`

- **@Metadata**
  `Reflective metadata object.`

- **@Metadata.property**
  `A property of a metadata object.`

- **@Metadata.property =** *value*
  `Assign a value to the metadata's property.`

- ~**s/**[^]*original_value*[$]**/***new_value***/**
  `Substitute` *`original_value`* `with` *`new_value`*`, as specified by regular expressions.`

- *program_construct_variable* **+= @Metadata**
  `Add @Metadata to` *`program_construct_variable`*`.`

Figure 10: PBSE constructs.

c denotes a class
m denotes a method
f denotes a field
p denotes a parameter
a denotes an annotation

$A_C(c)$ denotes the set of annotations of class c
$A_M(m)$ denotes the set of annotations of method m
$A_P(p)$ denotes the set of annotations of parameter p
$A_F(f)$ denotes the set of annotations of field f

P_c denotes a class regular expression pattern
P_m denotes a method regular expression pattern
P_p denotes a parameter regular expression pattern
P_f denotes a field regular expression pattern

$RE(e, P_e)$ denotes a regular expression match of a language construct e over pattern P_e

Figure 11: Syntax definitions.

$$\frac{RE(c,\ P_c)\ (P_c,\ a) \in PBSE}{a \in A_C(c)}\ [\ AnnotateClass\ ]$$

$$\frac{RE(m,\ P_m)\ (P_m,\ a) \in PBSE}{a \in A_M(m)}\ [\ AnnotateMethod\ ]$$

$$\frac{RE(p,\ P_p)\ (P_p,\ a) \in PBSE}{a \in A_P(p)}\ [\ AnnotateParameter\ ]$$

$$\frac{RE(f,\ P_f)\ (P_f,\ a) \in PBSE}{a \in A_F(f)}\ [\ AnnotateField\ ]$$

Figure 12: Translation rules.

constructs could be potentially annotated, and a set of available annotations appears as well. Each annotation is specific to the type of a program construct to which it can be added, including classes, methods, method parameters, and fields. The same annotation could potentially be used at multiple levels; for example, the `@Column` annotation can be applied to both methods and fields in the Java Persistence API.

Each structural program construct can be matched with a regular expression pattern, which are formed by replacing some substring of a construct with a wildcard character (e.g., `*`) that can match multiple constructs. The regular expressions of PBSE `Where` clauses have been inspired by AspectJ pointcuts [40]. Figure 12 uses set operations to show how various constructs are annotated if they are matched by the PBSE regular expressions. Specifically, an annotation a is added to the annotations of a program construct e (i.e., class, method, parameter, or field) precisely when the construct matches a regular expression pattern, and the annotation a is attached to that pattern in the PBSE specification.

The presented translation semantics is a simplification, in that it describes only the main translation rules from PBSE to annotations. More complex features of PBSE, including nested patterns, module application, and substitutions, have been elided to save space and streamline the presentation.

4. Enabling Cross-Language Metadata Reuse

In order to support the cross-language reusability, our approach requires that the source-to-source compilation mappings between the emerging and target languages be made available. NFC implementers (e.g., an ORM or a unit testing framework vendor) can then use these mappings to derive a simple declarative specification that expresses how to compile PBSE across languages.

To that end, our approach provides a simple declarative DSL that is derived from PBSE. The resulting PBSE mapping specification parameterizes a generator that synthesizes a PBSE cross-translator (Section 4.2).

4.1. Programming Model

Figure 13 gives an overview of our approach. To add an implementation of an NFC to a program in the source language, the programmer writes a PBSE specification that refers to that program's constructs. For each concern to be added, the programmer needs to provide a separate PBSE specification. For example, if a program needs both persistence and security, the program must specify separate PBSE specifications for each of these two NFCs. PBSEs are then translated from the source to the target language specifications. Our approach supports PBSE for multiple languages, including Java, C++, X10, Scala, and C#. Then, the PBSE specifications in the target language are translated to the metadata format required by the NFC's target language implementations. Each implementation may use a different metadata format and sometimes use multiple formats simultaneously. For example, the Java Data Objects persistence system takes as input both XML files and Java 5 annotations. At the same time, the Security Annotation Framework (SAF)[31] requires that the programmer use Java 5 annotations. Our approach can translate a PBSE

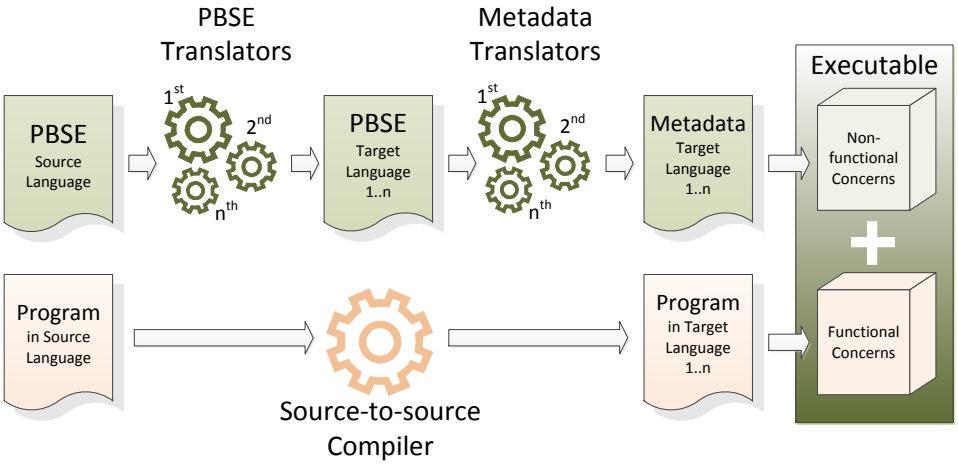


Figure 13: Generating Target Sources and Metadata formats.

specification to all the major metadata formats, including XML files, annotations, pragmas, and macros. Once the translated metadata is added to the program emitted by the source-to-source compiler, the resulting executable artifact implements both functional and non-functional concerns. The functional concerns are implemented by translating the source program to the target one, while the NFCs are implemented by adding the appropriate metadata to the target program.

This process must be repeated for each of the supported source-to-source compilation targets. For example, the X10 compiler emits both Java and C++. Thus, if an NFC is needed in both backends, the appropriate metadata has to be generated for each of them. Because language ecosystems tend to implement the same NFC distinctly, the NFC implementations in each compiled language may require different metadata formats and content. For example, for the persistence NFC, a Java ORM may require XML configuration files, while a C++ ORM may require C/C++ pragmas.

4.2. Cross-Language Metadata Translation

Our design is based on the assumption that if a source language can be source-to-source compiled to a target language, then the metadata with which a source language program is tagged can be translated to tag the resulting target language program. Figure 14 demonstrates this assumption pictorially. We assume that (1) the program's source-to-source compiler is not aware of metadata, and (2) the metadata's compiler can be derived from the program's source-to-source compiler. This entails that metadata is external to the source language.

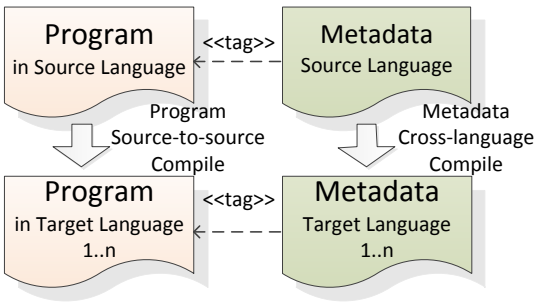


Figure 14: Translating Metadata formats.

As a general strategy, PBSE specifications are translated across languages. A PBSE specification for X10 is translated to PBSE specifications for Java and C++, whenever an X10 program is compiled to these languages. We call this translation process *cross-language metadata transformation*. In addition, PBSE specifications can be translated to mainstream metadata formats, including XML, Java 5 annotations as well as C/C++ pragmas and macros.

4.2.1. Metadata translation framework

To facilitate cross-language metadata transformation, our solution is two-pronged: metadata translation is specified declaratively and implemented using a generative approach. That is, to express metadata translation rules, our approach features a declarative domain-specific language. In addition, we provide a PBSE translation framework that transforms a PBSE specification into an abstract syntax tree that can be operated on using visitors. Our code generator takes declarative metadata translation rules and synthesizes the translation visitors.

4.2.2. PBSE meta-metadata

Within the same language, different metadata formats for a given NFC tag the same program constructs. Across languages, the tagged source language constructs map to their source-to-source compilation targets. Because NFC metadata tags structural program constructs (i.e., classes, methods, and fields), one can express declaratively how metadata is to be translated both within and across languages.

To that end, our approach extends PBSE with *meta-metadata*—meta constructs that codify differences between metadata formats. In Figure 15, we show meta-metadata for translating between PBSE for X10 (Figure 9) and Java (Figure 30). The right arrow operator (->) specifies a unidirectional transformation, while the double arrow (<->) specifies a bidirectional transformation. In essence, a meta-metadata script contains a declarative specification of transformations required to transform one metadata format to another. The declarative nature of meta-metadata facilitates the construction of metadata translators.

Because metadata applies to structural program constructs (i.e., classes, methods, and fields), meta-metadata needs to express how these structural constructs map to each other between the source and target languages. Meta-metadata specifications are to be crafted by language compiler writers—intimately familiar with how their source language translates into the target language—who can easily declare the mapping.

The meta-metadata in Figure 16 expresses how to translate from PBSE to XML for the JDO ORM. Pattern matching expresses how different metadata variables, depicted as Java 5 annotations, should map to the corresponding XML tags.

4.2.3. Generative Visitors

Based on the meta-metadata specification in Figure 16, our code generator synthesizes a visitor class shown in Figure 18. Because it would not be pragmatic to generate all code from scratch, the generated `PBSEVisitorJavaToXML` class references several classes provided as a library. In particular, it extends the `PBSEVisitorAdaptor` class and manipulates various PBSE AST element classes such as `PBSEElementClass` and `PBSEElementField`. It also uses a utility class `JavaToXML` that encapsulates low-level translation functionality. The XML in Figure 19 was produced by one of the generated visitors.

Figure 17 presents a UML diagram of the visitors used in the examples discussed above. All the core pieces of our translation framework have been implemented. Some of the code generation functionality is provided by code templates. Future work will refine our code generation infrastructure and explore whether some library pieces can be generated from scratch instead.

```
MetaMetadata PBSEX10toJava<PBSE pbse>
  Where (Class c in pbse)
  Where (public struct *)
    "struct" -> "class"
  Where (Field f in pbse)
  Where (private * ${temp1}:${temp2})
    ${temp1} <-> ${temp2}
  Where (private * *:*)
    ":" -> "\s"
  Where (private val *:*)
    "val" -> "final"
  Where (private var *:*)
    "var" -> ""
  Where (Method m in pbse)
  Where (* def *:${returntype})
    "def" -> ${returntype}
  ...
```

Figure 15: Meta-metadata for translating PBSE from X10 to Java.

```
MetaMetadata PBSEJavaToXML<PBSE pbse>
  Where (Class c in pbse)
    @Table -> "<class>"
      @Table.name ->
        "<class name=" + c.name + "/>"
      @Table.class ->
        "<class table=" + c.table + "/>"
    "</class>"

  Where (Field f in pbse)
    @Field -> "<field>"
      @Field.name ->
        "<field name=" + f.name + "/>"
      @Column -> "<column/>"
      @Column.name ->
        "<column name==" + f.column + "/>"
    "</field>"
```

Figure 16: Meta-metadata for translating PBSE for Java to XML used by the JDO system.

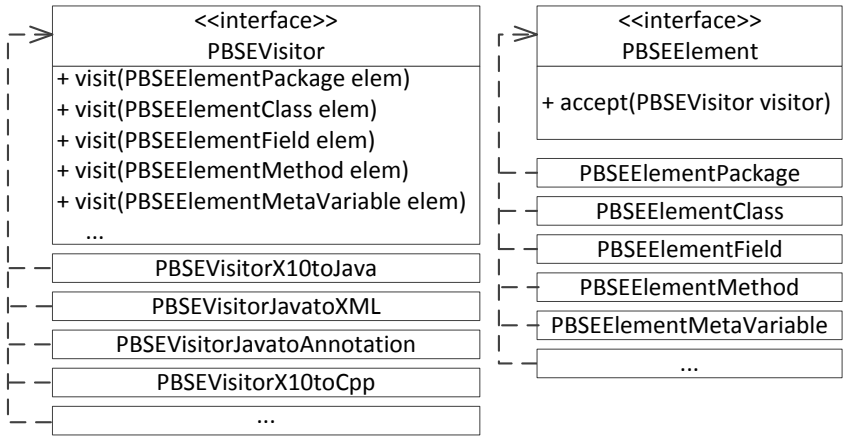


Figure 17: PBSE visitors translating metadata format.

```
class PBSEVisitorJavaToXML extends
  PBSEVisitorAdater {

  void visit(PBSEElementClass elem){
     if(elem.tagWith("@Table")){
       out.write(JavaToXML.
       translate("@Table","<class/>"));
     } else
     if(elem.tagWith("@Table.name")){
       out.write(JavaToXML.translate
       ("@Table.name",
        "<class table=${value}/>"));
     } else
     if(elem.tagWith("@Class.table")){
       out.write(JavaToXML.translate
       ("@Table.class",
        "<class name=${value}/>"));
  }}

  void visit(PBSEElementField elem){
     if(elem.tagWith("@Field")){
       out.write(JavaToXML.
       translate("@Field","<field/>"));
     } else
     if(elem.tagWith("@Field.name")){
       out.write(JavaToXML.translate
       ("@Field.name",
        "<field name=${value}/>"));
     } else
     if(elem.tagWith("@Column.name")){
       out.write(JavaToXML.translate
       ("@Column.name",
        "<column name=${value}/>"));
     } else
     if(elem.tagWith("@Column")){
       out.write(JavaToXML.translate
       ("@Column","<column/>"));
     }
    /*  other visit methods go here.  */
}}
```

Figure 18: A generated visitor.

```
<jdo><package name="ssca1">
<class name="SSCA1Model"
 table="SSCA1"
 identity-type="application">
 <field name="modelId" persistence-
  modifier="persistent" primary-key="true">
  <column name="MODELID"/>
 </field>
 <field name="winningScore"
  persistence-modifier="persistent">
  <column name="WINNINGSCORE"/>
 </field>
 <field name="shorterLast"
  persistence-modifier="persistent">
  <column name="SHORTERLAST"/>
 </field>
 <field name="longerLast"
  persistence-modifier="persistent">
  <column name="LONGERLAST"/>
 </field>
 <field name="longOffset"
  persistence-modifier="persistent">
  <column name="LONGOFFSET"/>
 </field>
 ...
</class></package></jdo>
```

Figure 19: Translated XML metadata for the JDO system.

4.2.4. Generating Client APIs

Since not all NFC functionality can be expressed via metadata, some client API must supplement the automatically translated metadata process described above. To that end, our infrastructure features NFC-specific client APIs. These APIs are invoked to access certain NFC functionalities explicitly. For example, persistent objects may need to be stored and retrieved from stable storage within a transactional context. In lieu of a transaction framework based on metadata, one may provide a code template to easily add transactional context to the persistence operations performed on any object. Figure 20 shows our code templates that can be used to add transactional support to persisting objects in Java and C++, shown in the left and right parts of the figure, respectively. The code templates are parameterized with the needed program construct names. The parameters are distinguished by their names, with the $ sign prefixing each parameter. For example, `$[Class.name]` expresses that this parameter should be substituted with the value of this variable in the configuration file, composed of key-value pairs. The `$iterator[..]` metavariable iterates over all fields or methods of a class.

```
$[Class.name] getPersistentObj
    ($[Class.name] param) {
  PersistenceManager pm =
    getPersistenceManager();
  $[Class.name] pobj =
    getObj(pm,"$[Class.name].class",param);
  Transaction tx = pm.currentTransaction();
  tx.begin();
  ...
  if (pobj == null) {
    pobj = new $[Class.name]($iterator
        [param.$[Class.field.name]]);
    pm.makePersistent(pobj);
  }
  tx.commit();
  return pobj;
}
```

```
ref<$[Class.name]> getPersistentObj
    (ref<$[Class.name]> param) {
  auto_ptr < database > db
    (create_database (argc, argv));
  $[Class.name]* pobj = param._val;
  transaction t(db->begin());
  ...
  if (checkNull(pobj)) {
    db->persist(*pobj);
  }
  t.commit();
  return param;
}
```

Figure 20: The code template for generating database transaction API for the Java (left) and C++ (right) backend.

5. Evaluation

We evaluate PBSE for its effect on reducing programming effort and for its general reusability. To show how PBSE enables systematic reuse, we applied it to two separate NFCs expressed in two different languages.

5.1. Adding Unit Testing and Persistence to X10

We applied PBSE to reuse four NFC implementations across two domains and two languages. We reused the JUnit [41] and CppUnit [42] testing frameworks, thereby adding unit testing capabilities to X10. We also reused Java Data Objects (JDO) [38] and ODB [39], Java and C++ ORM systems, thereby adding transparent persistence to X10 programs. In the following description, we detail our experiences with reusing these NFC implementations in X10.

5.1.1. Unit Testing X10 Programs

As is true for many emerging languages, no unit testing framework has yet been developed for X10. Although unit testing is an NFC, it is an integral part of widely used software development methodologies such as test-driven development (TDD) and extreme programming (XP). As a result, programmers following these methodologies in other languages are likely to miss unit testing support when programming in X10.

Consider the X10 class `Integrate` (Figure 21) that uses Gaussian quadrature to numerically integrate between two input parameters—the left and the right values. This class comes from a standard IBM X10 benchmark [43]. An area is computed by integrating its partial parts. For example, when computing the area with the start of *a* and the

```
1  public class Integrate {
2    static def computeArea(
3        left:double,
4        right:double) {
5      return recEval(
6        left,(left*left+1.0)*left,
7        right,(right*right+1.0)*right,0);
8    }
9
10   static def recEval(l:double,r:double,..)
11   {
12     // ..
13     finish{async{
14         expr1 = recEval(c,fc,r,fr,ar);
15       };
16       expr2 = recEval(l,fl,c,fc,al);
17     }
18     // ..
19     return expr1 + expr2;
20   }
21 }
```

Figure 21: An X10 Integrate class to be unit tested.

```
1  public class IntegrateTest {
2    var parm : double;
3    var expt : double;
4    var integrate : Integrate;
5
6    def init() {integrate = new Integrate();}
7    def finish() {integrate = null;}
8    def this(parm : double, expt : double) {
9      this.parm = parm;
10     this.expt = expt;
11   }
12   public def testComputeArea() {
13     val result =
14         integrate.computeArea(0, this.parm);
15     TUnit.assertEquals(this.expt, result);
16   }
17   public static def data() {
18     val parm = new Array[double](0..1*0..2);
19     parm(0, 0) = 2;
20     parm(0, 1) = 6.000000262757339;
21     parm(1, 0) = 4;
22     parm(1, 1) = 72.000000629253464;
23     parm(2, 0) = 6;
24     parm(2, 1) = 342.000001284044629;
25     return parm;
26 }}
```

Figure 22: The unit testing class for the Integrate class in Figure 21.

end of b, $\int_a^b f(x)dx$ computes partial results through integration. The application then sums up the partial integration results—$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{b-a}{2}x + \frac{b+a}{2})dx$.

Gaussian quadrature is non-trivial to implement correctly, but this implementation is even more complex as it does involve parallel processing. X10 `async` and `finish` constructs spawn and join parallel tasks, respectively. Even a testing skeptic would want to carefully verify a method whose logic is that complex. The irony of the situation is that both of the X10 compilation targets—Java and C++—have mature unit testing frameworks developed for them (e.g., JUnit and CppUnit [44]). The programmer should be able to write unit tests in X10 program, and depending on the source-to-source compilation target, compile these tests to be run by JUnit or CppUnit.

To implement and run unit tests in X10, the programmer first implements the needed unit tests in an X10 class. For example, the unit tests for class `Integrate` is shown in Figure 22. This class implements a typical test harness required by major unit testing frameworks. In particular, methods `init` and `finish` initialize and cleanup the test data, respectively. Method `testComputeArea` tests method `computeArea` in class `Integrate` by asserting that the method's result is what is expected. Method `data` provides the parameters for different instantiations of class `IntegrateTest` as a multidimensional array, in which each row contains a parameter/expected value pair, located in first and second columns, respectively.

To translate this code to work with unit testing implementations in Java and C++ as shown in Figure 23, the programmer also has to declare a simple metadata specification shown in Figure 24. This specification establishes a coding convention as the one used in class `IntegrateTest`. Notice how this metadata specification can be *reused* with all the classes ending with suffix "`Test`" in a given package.

Given this PBSE specification as input, our approach then generates the Java or C++ code required to run the translated test harness of the unit testing framework at hand. Through code generation, our approach can addresses the incongruity of features in different NFC implementations. While parameterized unit tests are supported by JUnit in the form of the `@RunWith(value=Parameterized.class)` annotation, CppUnit has no corresponding feature to implement this functionality (the left part of Figure 23). In addition, JUnit requires that the method providing the parameters for unit test instantiations return `java.util.Collection` (the right part of Figure 23). Unfortunately, the X10 method `data` is compiled to a Java method returning `x10.array.Array`, which does not extend this Java interface.

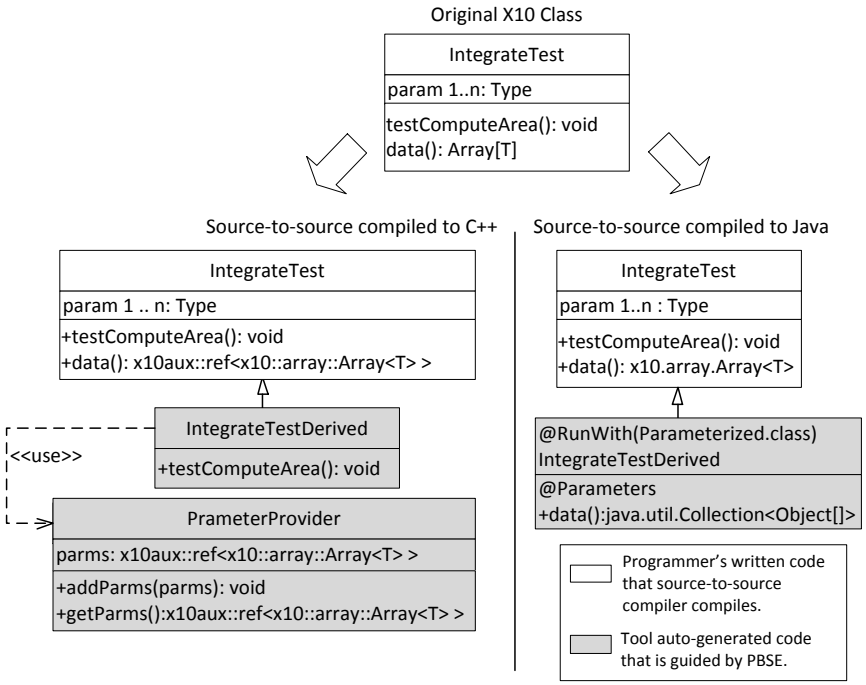


Figure 23: The class diagram for unit testing X10 programs with JUnit and CppUnit.

To overcome these limitations, our code generator synthesizes special classes that subclass the Java and C++ unit test classes emitted by the X10 compiler. These methods provide the required functionality by leveraging the Adaptor design pattern. To ensure that the `data` methods returns the required `java.util.Collection`, an adaptor method in the subclass invokes the base class method and wraps the returned type to an instance of `java.util.ArrayList`, thus satisfying this JUnit convention (Figure 26).

Supporting parameterized unit test execution in CppUnit requires more elaborate code generation. In particular, CppUnit features special macros to designate test classes and methods. C++ macros serve as predecessors of modern metadata formats such as XML files and annotations. The defining characteristic of framework metadata is the ability to express functionality declaratively, describing *what* needs to take place rather than *how* it should be accomplished.

```
1  Metadata UnitTest<Package p>
2    Class c in p
3    Where(public class *Test)
4      c += @RunWith
5      @RunWith.value = "Parameterized"
6      TestMethod<c>
7  Metadata TestMethod<Class c>
8    Method m in c
9    Where (public def init ())
10     m += @Before
11   Where (public def finish ())
12     m += @After
13   Where (public def test* ())
14     m += @Test
15   Where (public static def data ())
16     m += @Parameters
17 UnitTest<"integrate">
```

Figure 24: The PBSE specification for unit testing the X10 program.

```
1  void cppUnitMainTestSuite() {
2    INIT_TEST();
3    INIT_PARAMETER(ParameterProvider);
4    PARM_ITERATOR(SIZE()) {
5      ADD_TEST(IntegrateTest, // a test  class .
6               testComputeArea); // a  test  method.
7    }
8    RUN_TEST();
9  }
```

Figure 25: Extended macros based on CppUnit.

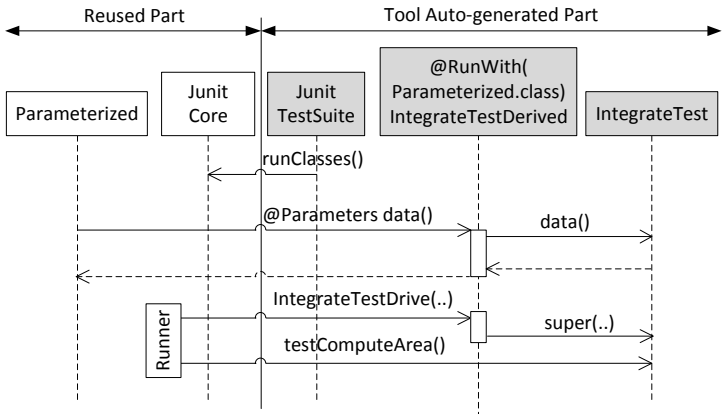


Figure 26: The sequence diagram for unit testing X10 programs with JUnit.

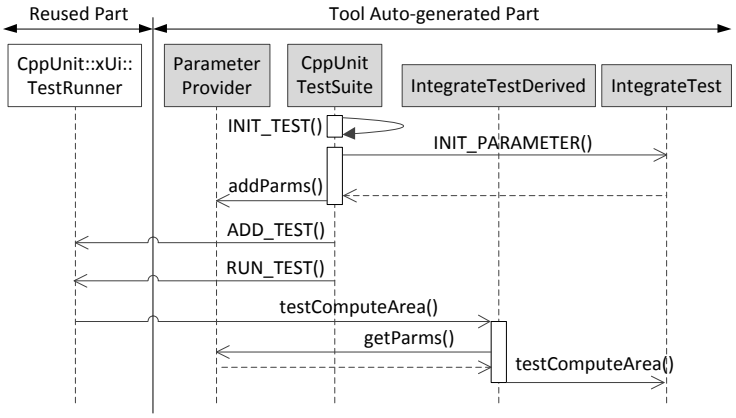


Figure 27: The sequence diagram for unit testing X10 programs with CppUnit.

In that regard, C/C++ macros are commonly used to define a DSL for expressing functionality at a higher abstraction level.

The macros in Figure 25 play the role of metadata that specifies how the CppUnit test harness should execute the tests defined in class `IntegrateTest`. To simplify the required metadata translation, we extended the built-in set of CppUnit macros to support parameterized unit tests. These macros are regenerated from scratch for every PBSE translation. The CppUnit macros express declarative metadata directives to initialize the framework, instantiate parameterized unit test classes, add them to a test harness, and run the added test methods (Figure 27).

Standard implementations of NFCs in richer languages expectedly provide more features and capabilities. In the case of unit testing, JUnit has built-in support for parameterized unit testing. As a result, adapting the X10 Java backend to work with JUnit is more straightforward than adapting the C++ backend for CppUnit. In particular, the `@RunWith` annotation is natively supported by JUnit. Thus, to annotate the Java methods returning the parameterized test parameters with `@RunWith`, they simply need to be adapted to return `java.util.Collection`, as discussed above.

5.1.2. Transparently Persisting X10 Programs

Next we describe how we applied our approach to enhance X10 programs with transparent persistence capabilities. Both source-to-source compilation targets of X10—Java and C++—feature ORM engines that can be reused to make X10 programs transparently persistent.

Figure 28 shows an X10 class `Fmm3d` that implements the Fast Multipole Method for electrostatic calculations with analytic expansions [37]. The implementation is real and current: it follows the strategy outlined by White and Head-

```
1  public class Fmm3d{
2    def getDirectEnergy() : Double{
3      val model = new FmmModel();
4      val directEnergy = finish (SumReducer()){
5        ateach (p1 in locallyEssentialTrees) {
6          var thisPlaceEnergy : Double = 0.0;
7          for ([x1,y1,z1] in lowestLevelBoxes.dist(here)){
8            val box1 = lowestLevelBoxes(x1,y1,z1) as FmmLeafBox;
9            for ([atomIndex1] in 0..(box1.atoms.size()-1)){
10             for (p in uList){
11               for ([otherBoxAtomIndex] in 0..(boxAtoms.size-1)){
12                 thisPlaceEnergy += atom1.charge*atom2Packed.charge /
13                   atom1.centre.distance(atom2Packed.centre);
14               }
15             }
16           }
17           model.setModelId(id(box1.x,box1.y,box1.z));
18           model.setEnergy(thisPlaceEnergy);
19           // ...
20           TP.setFmmModelObj(model);
21         }
22         offer thisPlaceEnergy;
23       }
24     };
25     return directEnergy;
26   }
27 }
```

Figure 28: A persisting class `Fmm3d` (simplified version) for the X10 `FmmModel` class in Figure 8.

Gordon [45] which was recently enhanced by Lashuk et al. [46]. The `getDirectEnergy` method sums the value of direct energy—`directEnergy`—on line 4 for all pairs of atoms in non-well-separated boxes. This operation requires only that atoms be already assigned to boxes, and can be executed in parallel with the other steps of the algorithm.

The ability to transparently persist a program's data can be used in multiple scenarios. For class `Fmm3d`, a programmer may want to optimize the execution by keeping a persistent cache of known values of `thisPlaceEnergy`. The cache must be persistent if different processes invoking the algorithm are to take advantage of it. The required functionality can be added to the program by using the PBSE specification from the motivating example (Figure 9). Based on this specification, our approach generates all the required metadata for the ORM system at hand, for either the Java or C++ backend, as well as X10 API through which the programmer can explicitly save and retrieve the persisted state. The generated X10 Application Programming Interface (API) that provides various platform-independent convenience methods for interfacing with the platform-specific implementations. The API is represented as a single X10 class, `TP` (short for **T**ransparent**P**ersistence). For example, to restart a program from a saved state, the X10 programmer can use the provided `TP` API class as follows: `val pobj = TP.getModel().getModelObj(latestCheckID)`. Our approach effectively addresses the incongruity of features in different NFC implementations.

In this case study, we reused two mainstream, commercial ORM systems for Java and C++, JDO and ODB. While JDO uses XML files or Java annotations as its metadata format, ODB uses C++ pragmas. Nevertheless, our approach seamlessly supported these disparate metadata formats, with the metadata for both Java and C++ backends automatically generated from the same PBSE X10 specification.

Figure 29 depicts a segment of the generated JDO XML deployment descriptor. To generate this deployment descriptor, our approach uses the PBSE depicted in Figure 30. Parameterized with this descriptor, the JDO runtime can transparently persist the specified X10 fields when the program is compiled to Java. Figure 31 depicts a segment of the generated ODB pragma definitions as described in the PBSE specification in Figure 32. Parameterized with a file containing these pragmas, the ODB compiler generates the functionality required to transparently persist the specified X10 fields when the program is compiled to C++. Both JDO and ODB can create a relational database table to store the transparently persistent state. Furthermore, both backends share the same database schema. In other words, if an X10 program is compiled to both Java and C++ backends, both of them will share a database schema and

```
<jdo>
<package name = "au.edu.anu.mm">
<class name = "FmmModel"
  table = "Fmm"
  identity-type = "application">
  <field name = "modelId"
    persistence-modifier = "persistent"
    primary-key = "true">
    <column name = "MODELID"/>
  </field>
  <field name = "energy"
    persistence-modifier = "persistent">
    <column name = "ENERGY"/>
  </field>
  ...
</class>
</package> </jdo>
```

Figure 29: Translated XML for the JDO system.

```
Metadata PersistentJava<Package p>
  Class c in p
  Where (public class *Model)
    c += @Table
    @Table.name = (c.name=~s/Model$///)
    Column<c>
Metadata Column<Class c>
  Field f in c
  Where (private * *)
    Method m in c
    Where((get+(f.name=~ s/^[a-z]/[A-Z]/)) == m.name)
      m += @Column
      @Column.name = (f.name=~s/[a-z]/[A-Z]/)
    Where (public * *Id ())
      @Column.primaryKey = true
      m += @Id
PersistentJava <"ssca1">
```

Figure 30: PBSE for transparent persistence in Java.

```
#ifndef ODB_MAPPING_H
#define ODB_MAPPING_H

#include <x10/lang/Runtime.h>
#include <x10aux/bootstrap.h>
#include <x10/lang/Runtime.h>
#include <x10aux/bootstrap.h>
#include "FmmModel.h"

#pragma db object(FmmModel) table("Fmm")

#pragma db member(FmmModel::FMGL(modelId))
   id column("MODELID")

#pragma db member(FmmModel::FMGL(energy))
   column("ENERGY")
...
#endif
```

Figure 31: Translated C++ pragmas for the ODB system.

```
Metadata PersistentCpp<Package p>
  Class c in p
  Where(class *Model)
    c += #pragma
    #pragma.object = c.name
    #pragma.table = (c.name =~ s/Model$///)
    Field<c>
Metadata Field<Class c>
  Field f in c
  Where (* *)
    Method m in c
    Where((get + (f.name =~ s/^[a-z]/[A-Z]/)) == m.name)
      f += #pragma
      #pragma.member = c.name + "::FMGL(" + f.name + ")"
      #pragma.column = (f.name =~ s/[a-z]/[A-Z]/)
    Where (* *Id ())
      #pragma.id = true
PersistentCpp<"model">
```

Figure 32: PBSE for transparent persistence in C++.

thus can interoperate with respect to their persistent state. If the Java backend persists its state, it can then be read by the C++ backend and vice versa.

In summary, Figure 33 illustrates how an X10 program can reuse NFCs implemented in Java and C++, the X10 source-to-source compilation targets. Thus far, we have evaluated the effectiveness of our approach on the ability to reuse unit testing and transparent persistence. However, our approach should be applicable to reusing other NFCs configured through metadata such as security, particularly with respect to encryption and access control.

5.2. Performance Evaluation

To evaluate the performance of the reused implementations of transparent persistence in Java and C++, we added checkpointing to X10 programs. Checkpointing periodically saves a long-running computation's intermediate results to stable storage that can be used to recover from failure. To avoid restarting from the beginning in case of a crash, the intermediate results are used to restart from the latest checkpoint.

To ensure high efficiency, checkpointing is commonly hand-crafted. In contrast, we checkpointed our benchmark programs through the added transparent persistence. Although transparent persistence may not be the most efficient way to checkpoint a program, it stress tests the performance of transparent persistence mechanisms. Thus, we measure the overhead of our checkpointing functionality rather than compare it to a hand-crafted solution.

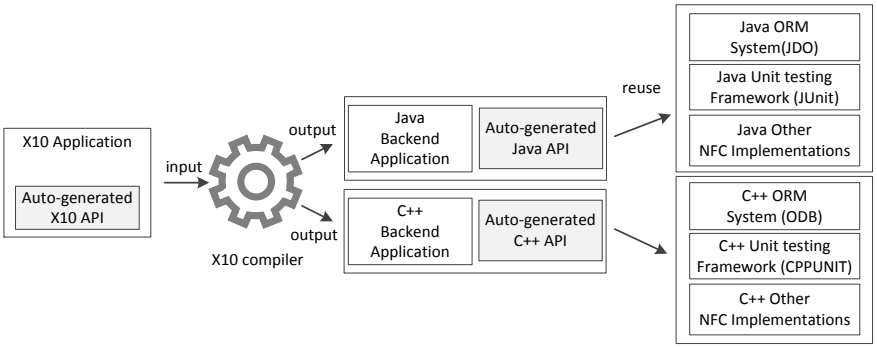


Figure 33: Reusing Java and C++ NFC implementations in X10.

We assess the efficiency and scalability of our transparent persistence implementation through micro and macro benchmarks. We added checkpointing to two third-party X10 applications and measured the overhead of our approach by benchmarking the checkpointing functionality. The measurements were performed on Linux version 2.6.32-30-generic, Dell Optiplex GX620, Intel Pentium CPU 3.00GHz, and 2.00 GB RAM. In all Java benchmarks, the rest of the setup consisted of Java Runtime build 1.6.0_21-b07, and JDO 2.2. In all C++ benchmarks, we used, g++ 4.4.3, and ODB 1.1.0. For all the benchmarks, we used X10 build 2.1.1 and MySQL 5.1.41, which provides high efficiency with a small resource footprint.

Our results indicate that our approach is efficient and scalable, with the checkpointing functionality incurring only a small percentage of the overall execution time, and the overhead being linearly correlated with the checkpointed state's size.

Checkpointing. In our experiments, we added checkpointing capabilities to two X10 third-party applications: (1) *Fmm3d* [37] Fast Multipole Method for electrostatic calculations and (2) *SSCA1* [47] the Smith-Waterman DNA sequence alignment algorithm [48]. SSCA1 computes the highest similarity scores by comparing in parallel an unknown sequence against a collection of known sequences. As discussed in Section 5.1.2, Fmm3d is Fast Multipole Method for distributed electrostatic calculations in a cubic simulation space centered at the origin.

In this benchmark, we compared the total execution time of the following two application versions: (1) without the persistence capability and (2) with the persistence capability. While the applications without the persistence capability kept the checkpointed data in memory, the applications with the persistence capability synchronized the checkpointed data in memory with a database. In our setup, we placed subject applications and the database on the same machine, so as to exclude network latency from our measurements. The purpose of this comparison was to verify that the added checkpointing functionality does not negatively affect program scalability. Specifically, for both benchmark applications, we measured the total execution time of the two versions, with the number of checkpoints increasing from 4 to 20 in the increments of 4. A unit of checkpointed data comprised an object with about a dozen primitive fields.

As the bottom graphs in Figure 34(a) and Figure 34(b), respectively, demonstrate, the checkpointing functionality implemented via transparent persistence neither incurs significant performance overhead nor hinders scalability. The incurred overhead remained constant for the C++ versions. In general, the C++ backend is known to outperform the Java one. In addition, we chose to use a standard JDO implementation rather than a specially optimized ORM system.

Migrating between Java and C++ backends. We also measured the total execution time of running the SSCA1 using both backends. In particular, the execution would start in Java, checkpoint its state, and then restart in C++. While the C++ backend is more efficient, the Java backend offer richer profiling facilities. As a result, an advantageous execution scenario is to collect some execution statistics when running the Java backend, and then restart the execution in C++ taking advantage of the collected statistics.

The left of Figure 35 shows the total execution time of different mixes of Java and C++ backends. For example, `30:70` means an X10 program executes 30% of its computational load in Java, checkpoint its state, and then completes the remaining load (70%) in C++. For the right part of Figure 35, we fixed the input size[6] to show how increasing the amount of work performed by the C++ backend can improve the overall performance.

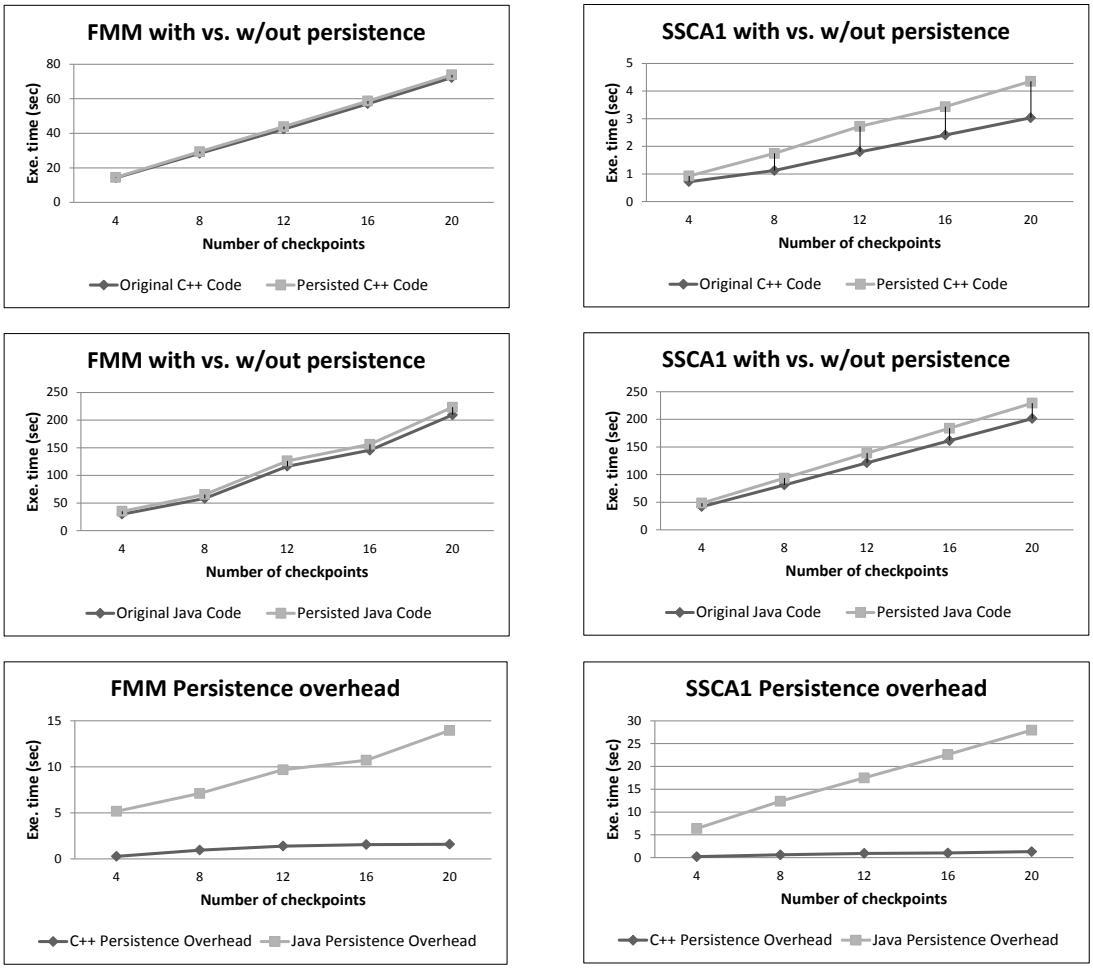
6. Discussion

Next, we compare PBSE with existing mainstream metadata formats. Then we discuss why PBSE enables effective cross-language reuse of NFCs.

6.1. Advantages

Compared to both XML and annotations, PBSE provides programmability, understandability, maintenance, and reusability advantages, summarized next.

[6]The input was fixed for the DNA sequences sizes 800K.



(a) FMM (b) SSCA1

Figure 34: The evaluation of the persistence overhead in the FMM (a) and SSCA1 (b) applications for both backends.

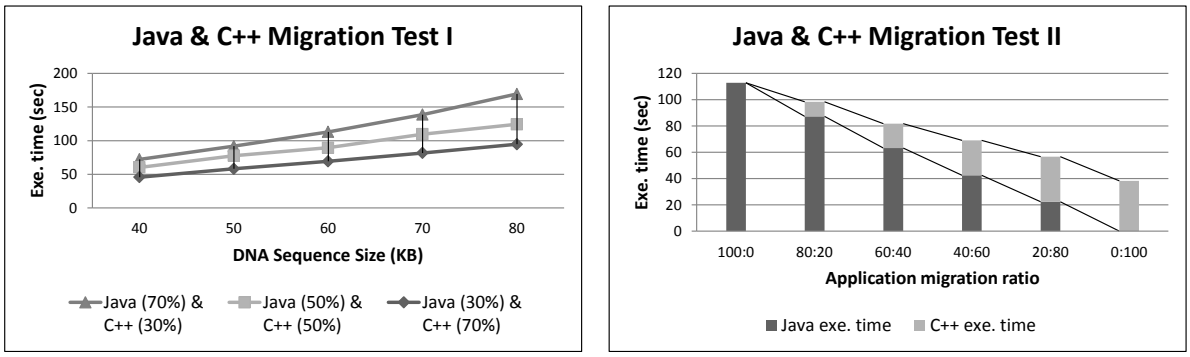


Figure 35: The performance test of application migration: execution time measurement migrating between Java and C++ backends compiled from a benchmark X10 application, SSCA1.

6.1.1. Programmability

PBSE expresses metadata concisely—a single pattern matches multiple program constructs, reducing the amount of metadata code that has to be written. We argue that PBSE will be fairly straightforward to learn for a developer familiar with object-oriented and declarative query (e.g., SQL) programming. Programming constructs such as iterators, `Where` clauses, and regular expressions, are part of the standard arsenal of a commercial software developer working with framework technologies. Finally, PBSE is more expressive than either XML or annotations, as they force the developer to understand and encode the structural correspondences between the main source code and its accompanying metadata.

6.1.2. Understandability

PBSE specifications capture the software architecture imposed by a given framework, something that neither XML or annotations can accomplish. XML was designed to facilitate the creation of automated parsers rather than to make XML documents easy to read and understand by a human user. In particular, the opening and closing XML tags often obfuscate the described data values and their relationship to one another. Annotations are easy to understand, but each individual annotation expresses only local information about its program construct; any correspondence between the name of an annotation and that of the program construct it annotates is only implicit. Any relationship between different program constructs annotated with the same annotation is only implicit also.

By contrast, by looking at a PBSE specification, programmers can understand the relationship between program constructs and their metadata. If the framework specific information can be encoded in PBSE specifications, programmers may not need to examine the source code—a short PBSE code snippet can capture complex naming conventions that hold throughout the entire codebase.

6.1.3. Maintainability

PBSE metadata alleviates the challenges of maintaining framework applications with respect to keeping the source code of an application consistent with metadata during program evolution. Although PBSE specifications are maintained separately from the main source code, their concise nature and the presence of explicit structural information make PBSE easier to maintain than either XML or annotations.

XML is disconnected from the main source code and must be evolved in parallel, thus doubling the maintenance programmer's burden. The location of the metadata information for a given program construct within an XML document is not immediately obvious, often taking a time consuming XML data exploration to discover. Because annotations remain next to the program construct they annotate, changing program construct names does not affect their annotations. However, newly added program constructs must also be annotated appropriately to ensure their proper interactions with the framework.

PBSE eliminates the Vendor-Lock In anti-pattern and preserves the correctness of metadata in the presence of program evolution, as long as a naming convention is followed. Switching to another framework to implement the same functionality is as straightforward as creating an alternative PBSE specification. A program can be enhanced with new methods, fields, and classes added or removed, and as long as the original naming convention is followed, no changes are required to keep the PBSE specification up to date. Having a PBSE specification to consult will make it less likely for the programmer to violate a naming convention when maintaining the code, as PBSE explicitly encodes the relationship between program constructs and metadata.

6.1.4. Reusability

Cross-component and -application reusability. PBSE metadata is reusable not only across different classes and packages within the same application, but also across different, and possibly unrelated, applications. By contrast, the existing metadata formats are not reusable. Neither XML or annotations are reusable, as they maintain a one-to-one relationship with the program constructs for which they provide metadata information. Annotations are particularly ill-equipped for reuse, as each program construct must be annotated individually.

The same PBSE metadata specifications can be reused in all the applications that use the same framework, as framework-dependent code is written to follow certain naming conventions. Furthermore, a PBSE specification can be easily modified for a different naming convention by changing only a few lines of code. For example, different prefixes or suffixes can be easily incorporated to accommodate the differences in naming conventions, and as long as the naming conventions are followed consistently, the slightly modified code can be fully reused.

Cross-language reusability. Due to their conciseness and simplicity, declarative metadata specifications are particularly amenable to automatic transformation, a property exploited by our approach.

Our approach would be inapplicable if NFCs were implemented through custom coding in mainstream languages. In fact, when reusing unit testing functionality, our approach addresses the issue of reusing test drivers and harnesses, facilities that execute programmer-written unit tests and report the results. Programmers still have to write their unit tests in X10, albeit using a provided assertion library.

Declarative approaches are widely used to implement the majority of NFCs. One reason for this is because aspect-oriented programming has entered the mainstream of industrial software development. Another reason is because metadata has been integrated into programming languages, such as Java 5 annotations and C# attributes. As declarative approaches become even more dominant, more functionality will become reusable through approaches similar to ours.

When applied to the same codebase, NFC implementations may harmfully interfere with each other. Although our approach does not change how NFCs are implemented, but only how they are expressed, we plan to explore whether PBSE can be extended with constructs that specify the order in which NFCs should be applied. When multiple NFCs influence the same program element, ensuring a specific order can help avoid some harmful interferences. Notice that mainstream metadata formats provide no such constructs.

So far, declarative abstractions have been used primarily to express NFCs. However, if portions of core functionality become expressible declaratively, the potential benefits of our approach will also increase. If metadata can be used to express some functional concerns, metadata translation can supplement or, in some instances, replace compilation.

6.2. Shortcomings

The advantages of PBSE are due to the structural patterns between the source code and metadata of modern framework applications. If, however, metadata is highly specialized for individual program constructs, without forming any patterns between the names of program constructs and their corresponding metadata, the utility and conciseness of PBSE can be compromised. For example, if every method or class in an application is annotated differently, the structural patterns of PBSE would not provide any conciseness or reusability advantages. One could still express such an application's metadata in PBSE, but each annotated program construct would require a separate `Where` PBSE clause. Similarly, using multiple frameworks may impose the need for multiple, different naming conventions. In that case, the complexity and/or number of `Where` PBSE clauses is likely to grow, thus potentially negating the format's software engineering advantages.

Some metadata-based frameworks do not maintain any logical relationship between metadata and tagged program constructs. For example, in OpenMP [49], a control predicate or a block statement can be tagged with `#pragma` (e.g.,

`#pragma omp parallel`) to specify a desired parallelization. In cases like this, PBSE would be inapplicable to reusing metadata.

Nevertheless, refactoring could increase the applicability of PBSE. For example, a refactoring approach in [50] extracts loop statements annotated with parallelization directives into individual methods. PBSE could then leverage the naming correspondences between the new methods and their annotations. One could even automatically suggest possible uses of PBSE by computing the program differences between the original and refactored versions [51]. The computed differences can then be used to generate PBSE scripts automatically.

The reliance on the naming conventions imposed by frameworks makes PBSE susceptible to what is known in AOP as the *fragile pointcut problem* [52]. Specifically, evolving a program can compromise the correctness of its PBSE declarations. However, the fragile pointcut problem of PBSE is alleviated by its reliance on the programming discipline imposed by frameworks.

As far as program evolution is concerned, program constructs could be added and removed, and their names could be changed. Removing a program construct or changing its name, within the confines of the naming convention in place, will not affect the correctness of PBSE declarations. If an added new construct is named according to a naming convention, it will be properly captured by PBSE. Annotations can make naming conventions optional. Take for example the difference between JUnit 3 and 4, the latter of which is annotation-based. With annotations, test methods no longer must start with "`test`", but can be named arbitrarily. One must ask, however, whether a test method's name should still contain the "`test`" substring. A naming convention that requires that a test method be named "`testFoo`" or "`fooTest`" improves readability, making the resulting code easier to understand and maintain, as it highlights the relationship between the unit testing annotations and the annotated program constructs. PBSE can capture such test methods irrespective of whether the naming convention in place uses "`test`" as the prefix or suffix.

Sometimes poorly-designed PBSE declarations can inadvertently capture program constructs that are not intended to be interacting with a framework. Consider expressing metadata as applicable to all getter methods and having non-getter methods, whose names start with "`get`"[7]. The PBSE examples introduced earlier would incorrectly capture such methods as getter. The PBSE declaration in Figure 36 avoids capturing such non-getter methods. Using a nested iteration over both fields and methods, this declaration defines a getter method as one whose name is a function of an existing private field's name.

```
1 Metadata Column<Class c>
2  Field f in c
3   Where (private * *)
4    Method m in c
5    Where(("get" + (f.name =~
6       s/^[a-z]/[A-Z]/)) == m.name)
7     m += @Column
8     @Column.name = f.name
9     ...
```

Figure 36: Identifying getters for `private` fields.

Although frameworks follow well-defined naming conventions not just with respect to program constructs but also to metadata, sometimes the naming conventions are broken inadvertently. To ensure that the utility of PBSE is not compromised, approaches to dealing with the fragile-pointcut problem in AOP, including delta analysis [52, 53] and pointcut rejuvenation [54], can be adopted.

Another promising approach to this problem is to control how programs evolve to avoid unsafety and inconsistencies. In a recent work, Abdelmeged et al. [55] propose that correctness criteria be declared explicitly and define a stricter notion of compatibility to identify inconsistencies. In the same vein, one could express the naming conventions of frameworks explicitly (e.g., using a language extension) and verify them using an automatic checker. One could also execute PBSE declarations as a query against the underlying program and examine the number of matched program constructs. A stricter notion of compatibility can be expressed in terms of the expected delta in the number of matches, in response to evolving the program. For example, a JUnit notion of compatibility, *N_Test_Methods == (N_Test_Methods + N_New_Test_Methods)*, can protect against the unsafe evolution resulting from mistakenly naming the newly added test methods as starting with "`tst`" rather than "`test`."

Finally, although Java packages can be annotated, package annotations are rare and typically are specified in a special file whose name is fixed to `package-info.java`. Based on these observations, we chose not to support structural expressions over package annotations in PBSE.

[7] `void getUpset(), void getAlarmed(), etc.`

7. Related Work

Our approach to reusing metadata across components, applications, and languages is related to a wide range of the state of the art in DSLs, pattern-matching, metadata validation/translation, expressing NFCs via AOP, and code generation. Next, we discuss some of this closely related state of the art.

Domain Specific Languages

Our approach hinges on the ability to translate metadata across different formats and languages. To express reusable cross-language metadata and how metadata is to be translated, we introduce new DSLs. DSLs have been widely used to streamline the implementation of various functionalities. *MetaBorg* [15] embeds domain-specific languages within a general purpose language, so that programmers can express domain abstractions using concrete syntax. For example, domain abstractions can be embedded within Java expressions. We could have used MetaBorg to embed PBSE within X10 expressions. Our design choice of placing PBSE in standalone source files, however, enables effective metadata reuse across components and applications. Douence et al. [56] present an AOP-based DSL to express crosscutting definitions (i.e., crosscutting operations) of NFCs. The DSL expresses patterns that monitor and detect events that describe callers, receivers, method names, arguments, results, and timestamps. Similarly to their approach, PBSE features patterns to match program constructs. Because NFCs are now commonly specified declaratively, PBSE only features static pointcuts that match such declarative specifications.

OptiML [57] is a DSL for machine learning applications, which bridges the gap of supporting applications on different platforms. The OptiML interpreter translates OptiML scripts to an intermediate representation, embedded in applications written in Scala, C++, and CUDA. Similar to OptiML, PBSE is a DSL for cross-platform metadata. By translating PBSE to different existing metadata formats (e.g, XML, annotations, macros, and #pragmas), existing NFCs can be reused from applications translated to different target languages.

FIISL [58] is a DSL for specifying how to integrate distributed applications running on different platforms. It is interpreted to generate interface program modules for communication between applications and platforms. Similarly, PBSE specification can be reused for different applications and components; however PBSE primarily leverages relationships between programs and metadata in application frameworks.

Metadata

Other more expressive metadata representations have been proposed in the literature. RDF [59]—an XML based metadata representation—improves network-based services such as the discovery and rating of resources. RDF associates values with properties and resources through a metadata schema. RDF provides flexibility and robustness advantages but is inapplicable to enterprise frameworks. SGF [60]—an XML based metadata representation—describes the structure of a web site to ease its navigation by creating interactive site maps; SGF also captures the semantic relationship between different web pages. The KNOWLEDGE GRID metadata [61] uses XML to represent information for managing the resources of a heterogeneous Grid, including computers, data, telecommunication, networks, and software. Orso, Harrold, and Rosenblum [62] discuss how metadata can be used to support a wide range of software engineering tasks with respect to distributed component-based systems. A particular focus of their work is testing and analysis of components.

Masmoudi, Paquette, and Champagne [63] introduce SOCOM, a metadata format that enables component aggregation and reuse. SOCOM is automatically translated into different component metadata formats, including XML, OWL, and relational schemas. While SOCOM was designed to facilitate component reuse, PBSE makes it possible to reuse metadata across components.

These metadata formats are quite useful to manage domain ontology using an expressive vocabulary to represent the types, properties, and relationship. PBSE could simplify the analysis of these metadata formats and models by capturing these architectural properties for framework-based applications.

Pattern-Matching Techniques

In the domain of XML processing, techniques have been proposed [64, 65, 66, 67] to extract general XML programming patterns. However, only the patterns that occur in XML files are considered, not the ones that codify the relationship between XML metadata and the program constructs it tags. Pattern-based reflective declaration [68, 69, 70]

is a meta-programming technique for generating well-typed program constructs such as classes, methods, and fields. This C# and Java language extension makes it possible to declare program fields and methods as a static, pattern-based, reflective iteration over other classes. Similarly, PBSE uses patterns over the structure of a program to express metadata. The programming languages community has proposed extending Java to enable the programmer to express pattern-matching [71, 72, 73]. These extensions describe how program constructs are declared and how they can be extracted based on the specified patterns. In addition, the declarative mechanism leverages the pattern-matching facility to add new functionality to existing one at the source or intermediate code levels. In some sense, PBSE extends the notion of pattern-matching facilities to enable systematic metadata reuse.

Validation for Metadata

PBSE is related to several research efforts whose objective is to validate the correctness of metadata. Eichberg et al. [74] check the correctness of annotation-based applications, in terms of their implementation restrictions and dependencies that are implied by annotations. The cases when the checked source code violates such restrictions and dependencies are reported by an automated, user-extensible tool. Noguera et al. [75] check the correctness of using annotations by adding to their declarations meta-annotations that define various constraints. Expressed as Object Constraint Language queries, these constraints must be satisfied when the declared annotations are used in the program. The definitions of annotation model constraints are validated at compile time by an automated tool. Cepa et al. [23] check the correctness of using custom attributes in .NET by providing meta-attributes that define dependencies between attributes. The attribute dependencies are expressed declaratively as a custom attribute and are checked using an automated tool. These approaches are quite powerful and can catch many inconsistencies of using metadata. The need for these approaches, however, is a testament that the mainstream metadata formats, such as Java annotations or .NET attributes, are not sufficiently expressive. This is the problem that PBSE aims to address by encoding the structural dependencies between program constructs and their corresponding metadata. By encoding such correspondences explicitly, PBSE specifications are less likely to contain unexpected inconsistencies and bugs. Furthermore, the declarative nature of PBSE makes it easier to ascertain complex metadata coding conventions by examining a single PBSE specification.

Metadata Translation

Similarly to our approach, several prior approaches also leverage metadata translation, albeit not across languages. Godby et al. [76] translate among the common metadata schemas by using syntactic transformation and semantic mapping to retrieve and create heterogeneous databases in the digital library's web service. MiningMart [77] presents a metadata compiler for preprocessing their metadata *M4* to generate SQL code while providing high-level query descriptions for very large databases. Ruotsalo et al. [78] transform across different metadata formats to achieve knowledge representation compatibility in different domains by means of domain knowledge. Hernández et al. [79] translate their custom metadata specifications for database mapping and queries. Popa et al. [80] generate a set of logical mappings between source and target metadata formats, as well as translation queries while preserving semantic relationships and consistent translations, focusing on capturing the relationship between data/metadata and metadata/data translations. These metadata translation approaches are quite powerful and can avoid inconsistencies when translating metadata. Our approach follows similar design principles but focuses on systematic metadata reuse; it also provides meta-metadata to encode the translation rules. The objective of our approach is to leverage the power of metadata translation to seamlessly reuse NFCs across components, applications, and languages.

Reusing Non-Functional Concerns with AOP

Aspect-oriented Programming [81] is the foremost programming discipline for implementing NFCs. Which NFCs are amenable to separate treatment has been the subject of an ongoing debate [82]. However, our approach reuses only those NFCs that are already expressed separately. Even though our approach does not use any mainstream AOP tools, it follows the general AOP design philosophy of treating cross-cutting concerns separately and modularly. AOP tools, including AspectJ 5 [83] and JBoss AOP [84], can introduce metadata to programs (e.g., `declare annotation` and `annotation introduction`), thereby implementing NFCs. However, these means of introducing metadata are not easily reusable as they are not parameterizable. As compared to AspectJ 5 and JBoss AOP, PBSE captures the structural correspondences between program constructs and metadata, and as a function of the program constructs can be reused across multiple programs.

Code Generation

Much of the effectiveness of our approach is due to its heavy reliance on template-based code generation. The benefits of this technique are well-known in different domains. Milosavljević et al. [85] map Java classes to database schemas by generating database code given an XML descriptor. XML schema elements translate to Java classes, fields, and methods. Our approach relies on standardized, mainstream implementations of NFCs. Instead of generating database code directly, our approach generates metadata that enables the target program to interface with platform-specific ORM systems. *DART* [86] is an automated testing technique that uses program analysis to generate test harness code, test drivers, and test input to dynamically analyze programs executing along alternative program paths. Based on an external description, the generated test harness systematically explores all feasible program paths by using path constraints. Our approach to reusing unit testing is similar in employing an external specification to describe tests. However, the X10 programmer still writes test harness code by hand. As future work, we may explore whether our approach can be integrated with a unit test generator such as JCrasher [87]. Devadithya et al. [88] add reflection to C++ by adding metadata to the compiled C++ binaries. Metadata classes are generated by parsing input C++ class and traversing the resulting syntax trees. Our approach to reusing NFCs across languages can be thought of as a cross-platform reflection mechanism, albeit limited to the program constructs interfacing with NFC implementations. Although our reflective capabilities are not as powerful and general, we support both Java and C++ as our target languages.

8. Future work and conclusions

The applicability of PBSE is not limited to Java, C++, and X10 programs only. This reusable metadata format can be applied to other languages that use either with built-in or external metadata. PBSE can add the metadata reusability benefits for applications written in emerging languages, including Scala, JRuby, and Jython, as an alternative to XML configuration files. PBSE can be used not only to configure frameworks, but to parameterize code generators and transformers.

In this article, we have presented Pattern-Based Structural Expressions, a new metadata format that enables systematic reuse. In addition, PBSE offers other software engineering benefits, as compared to mainstream metadata. PBSE leverages the coding conventions between the source code and its metadata commonly followed by modern enterprise applications. By explicitly capturing and expressing these patterns, PBSE specifications convey metadata information concisely and can be systematically reused.

To increase the applicability of PBSE, we have implemented a metadata translation framework that can produce the required metadata format given a PBSE specification. This framework can also add the needed metadata to existing applications as a one time task. The ability to automatically translate metadata for large codebases reduces the maintenance burden.

To validate the applicability of PBSE to emerging languages, we applied PBSE to reuse NFCs across languages, using X10 as our experimental target. Our approach supplements source-to-source compilation, so that emerging language programmers can take advantage of target language NFCs. As a specific application, we added unit testing and transparent persistence to X10 programs, thereby reusing four existing, mainstream, NFCs implementations in Java and C++. By eliminating the need to reimplement NFCs in emerging languages, our approach saves development effort.

As software development has become heavily dependent on the declarative model for implementing most of the NFCs, the role of metadata has gained prominence. Programmers spend a substantial amount of their time and efforts writing and modifying metadata. As a result, systematic metadata reuse is likely to yield tangible software engineering benefits. This work examines why mainstream metadata is not amenable to reuse and proposes an alternative metadata format that can be systematically reused across components, applications, and languages.

Acknowledgments

This research was supported by the National Science Foundation through the Grant CCF-111656 and by IBM through an X10 Innovation Award [89].

Availability

All the software described in the article is available from: `http://research.cs.vt.edu/vtspaces/x10pbse/`.

References

[1] W. B. Frakes, K. Kang, Software reuse research: Status and future, IEEE Transactions on Software Engineering (2005) 529–536.
[2] K. Schmid, A comprehensive product line scoping approach and its validation, in: Proceedings of the International Conference on Software Engineering, (2002) 593–603.
[3] G. Caldiera, V. R. Basili, Identifying and qualifying reusable software components, IEEE Computer (1991) 61–70.
[4] T. Ravichandran, M. A. Rothenberger, Software reuse strategies and component markets, Commun. ACM (2003) 109–114.
[5] J. S. Poulin, J. M. Caruso, D. R. Hancock, The business case for software reuse, IBM Systems Journal (1993) 567–594.
[6] T. Ravichandran, Special issue on component-based software development, SIGMIS Database (2003) 45–46.
[7] D. Batory, S. O'Malley, The design and implementation of hierarchical software systems with reusable components, ACM Trans. Softw. Eng. Methodol. (1992) 355–398.
[8] B. Burton, R. Aragon, S. Bailey, K. Koehler, L. Mayes, The reusable software library, Software, IEEE (1987) 25–33.
[9] M. L. Griss, Software reuse: From library to factory, IBM Systems Journal (1993) 548–566.
[10] S. Haefliger, G. von Krogh, S. Spaeth, Code reuse in open source software, Management Science (2008) 180–193.
[11] J. White, J. Hill, J. Gray, S. Tambe, A. Gokhale, D. Schmidt, Improving domain-specific language reuse with software product line techniques, Software, IEEE (2009) 47–53.
[12] D. Duggan, A mixin-based, semantics-based approach to reusing domain-specific programming languages, in: Proceedings of the European Conference on Object-Oriented Programming (2000) 179–200.
[13] T. Cleenewerck, K. Czarnecki, J. Striegnitz, M. Völter, Evolution and reuse of language specifications for DSLs, in: Proceedings of the European Conference on Object-Oriented Programming, ECOOP Workshops (2004), Springer.
[14] Oracle, Java platform enterprise edition, `http://www.oracle.com/technetwork/java/javaee/documentation/`.
[15] M. Bravenboer, E. Visser, Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2004) 365–383.
[16] K. Fisher, R. Gruber, PADS: A domain-specific language for processing ad hoc data, in: Proceedings of the Conference on Programming Language Design and Implementation (2005) 295–304.
[17] L. DeMichiel, JSR 153: Enterprise JavaBeans 2.1, `http://www.jcp.org/en/jsr/detail?id=153`.
[18] M. Vatkina, JSR 318: Enterprise JavaBeans 3.1, `http://jcp.org/en/jsr/detail?id=318`.
[19] S. Roock, A. Havenstein, Refactoring tags for automatic refactoring of framework dependent applications, in: Proceedings of the International Conference eXtreme Programming and Flexible Processes in Software Engineering (2002).
[20] T. Tourwé, T. Mens, Automated support for framework-based software evolution, in: Proceedings of the International Conference on Software Maintenance (2003).
[21] J. H. Perkins, Automatically generating refactorings to support API evolution, in: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (2005) 111–114.
[22] The Hibernate Team, Hibernate - relational persistence for idiomatic Java (2013), `http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/`.
[23] V. Cepa, M. Mezini, Declaring and enforcing dependencies between .NET custom attributes, in: Proceedings of the International Conference on Generative Programming and Component Engineering (2004) 319–331.
[24] C. Noguera, A. Kellens, C. D. Roover, V. Jonckers, Refactoring in the presence of annotations, in: Proceedings of the International Conference on Software Maintenance (2012) 337–346.
[25] M. Song, E. Tilevich, Metadata Invariants: Checking and inferring metadata coding conventions, in: Proceedings of the International Conference on Software Engineering (2012) 694–704.
[26] M. Inc., Java language conversion assistant (JLCA), `http://msdn.microsoft.com/en-us/magazine/cc163422.aspx`.
[27] J. W. Anderson, P. Lawson, M. Renschler, M. Lange, csUnit: The unit testing framework for the Microsoft .NET framework, `http://www.csunit.org/about.html`.
[28] V. Massol, T. Husted, JUnit in Action (2004).
[29] C. Beust, H. Suleiman, Next generation Java testing: TestNG and advanced concepts (2007).
[30] W. Brown, R. Malveau, H. McCormick III, T. Mowbray, AntiPatterns: Refactoring software, architectures, and projects in crisis (1998).
[31] The Maven Project Team, Security annotation framework, `http://safr.sourceforge.net/`.
[32] The Spring Project Team, Java web service, `http://static.springsource.org/spring-ws/docs/`.
[33] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, An overview of the scala programming language, LAMP-EPFL (2004).
[34] O. B. Charles Nutter, Thomas Enebo, N. Sieger, JRuby: The ruby programming on the JVM, `http://www.jruby.org/`.
[35] The Jython Project Team, Jython: Python for the Java platform, `http://www.jython.org/`.
[36] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, D. Grove, X10 language specification version 2.1 (2011), IBM Research.
[37] J. Milthorpe, V. Ganesh, A. P. Rendell, D. Grove, X10 as a parallel language for scientific computation: Practice and experience, in: Proceedings of the International Parallel & Distributed Processing Symposium (2011).
[38] The Oracle JDO Team, Java Data Objects (JDO), `http://www.oracle.com/technetwork/java/index-jsp-135919.html` (2012).
[39] Code Synthesis Tools CC Inc., ODB: C++ object-relational mapping, `http://www.codesynthesis.com/products/odb/`.
[40] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: Proceedings of the 15^{th} European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, London, UK, 2001, pp. 327–353.

[41] The JUnit Team, JUnit, https://github.com/junit-team/junit/ (2012).
[42] The CppUnit Team, CppUnit, http://sourceforge.net/projects/cppunit/ (2009).
[43] The X10 Team, IBM Research, Numerical integration implementation for gaussian quadrature, http://x10.svn.sourceforge.net/viewvc/x10/benchmarks/trunk/microbenchmarks/Integrate/.
[44] P. Hamill, Unit test frameworks (2004), O'Reilly.
[45] C. A. White, M. Head-Gordon, Derivation and efficient implementation of the fast multipole method, The Journal of Chemical Physics (1994) 6593–6605.
[46] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, G. Biros, A massively parallel adaptive fast-multipole method on heterogeneous architectures, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (2009) 58:1–58:12.
[47] The X10 Team, IBM Research, Smith-waterman algorithm implementation, http://x10.svn.sourceforge.net/viewvc/x10/benchmarks/trunk/SSCA1/.
[48] T. F. Smith, M. S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology (1981) 195–197.
[49] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, Computational Science & Engineering, IEEE 5 (1) (1998) 46–55.
[50] J. Sousa, J. L. Sobral, Jppal: Java parallel programming annotation library, in: Proceedings of the 2010 AOSD workshop on Domain-specific aspect languages, Vol. 3, 2010.
[51] B. Fluri, M. Wursch, M. PInzger, H. C. Gall, Change distilling: Tree differencing for fine-grained source code change extraction, Software Engineering, IEEE Transactions on 33 (11) (2007) 725–743.
[52] C. Koppen, M. Stoerzer, PCDiff: Attacking the fragile pointcut problem, in: European Interactive Workshop on Aspects in Software (2004).
[53] M. Stoerzer, J. Graf, Using pointcut delta analysis to support evolution of aspect-oriented software, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (2005) 653–656.
[54] R. Khatchadourian, P. Greenwood, A. Rashid, G. Xu, Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software, in: International Conference on Automated Software Engineering (2009).
[55] A. Abdelmeged, T. Skotiniotis, K. J. Lieberherr, Controlled evolution of adaptive programs, in: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (2009) 89–98.
[56] R. Douence, O. Motelet, M. Südholt, A formal definition of crosscuts, in: Proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (2001) 170–186.
[57] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, K. Olukotun, OptiML: an implicitly parallel domain-specific language for machine learning, in: Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011, pp. 609–616.
[58] J.-M. Lin, Cross-platform software reuse by functional integration approach, in: Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International, IEEE, 1997, pp. 402–408.
[59] E. Miller, An introduction to the resource description framework, Journal of Library Administration 34 (3) (2001) 245–255.
[60] O. Liechti, M. Sifer, T. Ichikawa, Structured graph format: XML metadata for describing Web site structure, Computer Networks and ISDN Systems 30 (1-7) (1998) 11–21.
[61] C. Mastroianni, D. Talia, P. Trunfio, Managing heterogeneous resources in data mining applications on grids using XML-based metadata, in: Parallel and Distributed Processing Symposium, 2003. Proceedings. International, 2003, p. 11.
[62] A. Orso, M. Harrold, D. Rosenblum, Component metadata for software engineering tasks, in: 2nd Int. Workshop on Engineering Distributed Objects (EDO 2000), Springer.
[63] A. Masmoudi, G. Paquette, R. Champagne, Metadata-driven software components aggregation process with reuse, International Journal of Advanced Media and Communication 2 (1) (2008) 35–58.
[64] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, W3C, XQuery 1.0: An XML Query Language (2007).
[65] V. Benzaken, G. Castagna, A. Frisch, CDuce: An XML-centric general-purpose language, in: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (2003) 51–63.
[66] H. Hosoya, B. Pierce, Regular expression pattern matching for XML, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (2001) 67–80.
[67] H. Hosoya, B. C. Pierce, XDuce: A statically typed XML processing language, ACM Trans. Internet Technol. (2003) 117–148.
[68] M. Fähndrich, M. Carbin, J. R. Larus, Reflective program generation with patterns, in: Proceedings of the International Conference on Generative Programming and Component Engineering (2006) 275–284.
[69] S. S. Huang, D. Zook, Y. Smaragdakis, Morphing: Safely shaping a class in the image of others, in: Proceedings of the European Conference on Object-Oriented Programming (2007) 399–424.
[70] S. S. Huang, Y. Smaragdakis, Class morphing: Expressive and safe static reflection, in: Proceedings of Conference on Programming Language Design and Implementation (2008) 79–89.
[71] T. Millstein, C. Frost, J. Ryder, A. Warth, Expressive and modular predicate dispatch for Java, ACM Trans. Program. Lang. Syst. 31 (2009) 7:1–7:54.
[72] K. Lee, A. LaMarca, C. Chambers, Hydroj: Object-oriented pattern matching for evolvable distributed systems, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (2003) 205–223.
[73] J. Liu, A. C. Myers, JMatch: Iterable abstract pattern matching for Java, in: Proceedings of the International Symposium on Practical Aspects of Declarative Languages (2003) 110–127.
[74] M. Eichberg, T. Schäfer, M. Mezini, Using annotations to check structural properties of classes, in: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (2005) 237–252.
[75] C. Noguera, L. Duchien, Annotation framework validation using domain models, Lecture Notes in Computer Science 5095 (2008) 48–62.
[76] C. J. Godby, D. Smith, E. Childress, Two paths to interoperable metadata, in: Proceedings of the International Conference on Dublin Core and metadata applications (2003) 1–9.

[77] K. Morik, M. Scholz, The miningmart approach to knowledge discovery in databases, in: In Ning Zhong and Jiming Liu, editors, Intelligent Technologies for Information Analysis (2003) 47–65.
[78] T. Ruotsalo, E. Hyvönen, An event-based approach for semantic metadata interoperability, in: Proceedings of the International Conference of Semantic Web (2007) 409–422.
[79] M. A. Hernández, P. Papotti, W.-C. Tan, Data exchange with data-metadata translations, Proceedings of the VLDB Endowment (2008) 260–273.
[80] L. Popa, Y. Velegrakis, M. A. Hernández, R. J. Miller, R. Fagin, Translating web data, in: Proceedings of the International Conference on Very Large Data Bases (2002) 598–609.
[81] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, J. Irwing, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming (1997).
[82] J. Kienzle, R. Guerraoui, AOP: Does it make sense? the case of concurrency and failures, in: Proceedings of the European Conference on Object-Oriented Programming (2002) 37–61.
[83] The AspectJ Project Team, The AspectJ 5 development kit developer's notebook, `http://eclipse.org/aspectj/doc/released/adk15notebook/index.html`.
[84] The JBoss Project Team, JBoss AOP: Java aspected oriented framework, `http://www.jboss.org/jbossaop/`.
[85] B. Milosavljević, M. Vidaković, Z. Konjović, Automatic code generation for database-oriented web applications, in: Proceedings of the Workshop on Intermediate Representation Engineering for Virtual Machines (2002) 59–64.
[86] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (2005) 213–223.
[87] C. Csallner, Y. Smaragdakis, JCrasher: An automatic robustness tester for Java, Software—Practice & Experience (2004) 1025–1050.
[88] T. Devadithya, K. Chiu, W. Lu, C++ reflection for high performance problem solving environments, in: Proceedings of the Spring Simulation MultiConference (2007) 435–440.
[89] E. Tilevich, X10 Innovation Awards: Automatic adaptation of Java frameworks for X10 to improve programmer productivity, `http://x10.codehaus.org/X10+Innovation+Awards`.