Accepted to AOSD 2012

Reusing Non-Functional Concerns Across Languages

Myoungkyu Song and Eli Tilevich

Dept. of Computer Science
Virginia Tech
{mksong,tilevich}@cs.vt.edu

Abstract

Emerging languages are often source-to-source compiled to mainstream ones, which offer standardized, fine-tuned implementations of non-functional concerns (NFCs)—including persistence, security, transactions, and testing. Because these NFCs are specified through metadata such as XML configuration files, compiling an emerging language to a mainstream one does not include NFC implementations. Unable to access the mainstream language's NFC implementations, emerging language programmers waste development effort reimplementing NFCs. In this paper, we present a novel approach to reusing NFC implementations across languages by automatically translating metadata. To add an NFC to an emerging language program, the programmer declares metadata, which is then translated to reuse the specified NFC implementation in the source-to-source compiled mainstream target language program. By automatically translating metadata, our approach eliminates the need to reimplement NFCs in the emerging language. As a validation, we add unit testing and transparent persistence to X10 by reusing implementations of these NFCs in Java and C++, the X10 backend compilation targets. The reused persistence NFC is efficient and scalable, making it possible to checkpoint and migrate processes, as demonstrated through experiments with third-party X10 programs. These results indicate that our approach can effectively reuse NFC implementations across languages, thus saving development effort.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Object-oriented programming; D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks; D.3.4 [*Programming Languages*]: Processors—Code generation, Interpreters, Parsing

General Terms Languages, Design, Experimentation

Keywords X10, Java, C++, source-to-source compilation, metadata, enterprise applications, non-functional concerns, transparent persistence, unit testing

1. Introduction

Modern industrial scale programming languages are much more than a grammar and syntactic rules for the programmer to follow. Mainstream enterprise languages feature complex and elaborate ecosystems of libraries and frameworks that provide standard application building blocks. In particular, many NFCs, including persistence, security, transactions, and testing, have been implemented in a standardized and reusable fashion. These implementations have become indispensable in modern enterprise applications. Examples abound: transparent persistence mechanisms facilitate data management; security frameworks provide access control and encryption; unit testing frameworks provide abstractions for implementing and executing unit tests, etc.

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 [19], JRuby [2], Jython [11], and X10 [22].

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.

In this paper, we present a novel approach to reusing NFC implementations across languages. Rather then 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 thesis behind our work is that it is possible to translate metadata alongside compiling the source language. Our approach requires expressive languages to specify metadata and how metadata is to be translated. We show how our Pattern Based Structural Expressions (PBSE) language [24] and its pattern-based implementation mechanism can play that role. For this work, we have extended PBSE to compile across languages, as specified by declarative translation strategies, to work with target language programs.

We validate the efficiency and expressiveness of our approach by adding unit testing and transparent persistence to X10, an emerging language being developed at IBM Research. The X10 compiler compiles an X10 program to both Java and C++, but does not implement unit testing or transparent persistence natively. We have reused well-known Java and C++ implementations of these NFCs in third-party X10 programs. X10 programmers express an NFC in PBSE, which is automatically translated to the metadata required for the NFC implementations in Java and C++.

Based on our results, this paper contributes:

- An approach to reusing NFC 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;
- The ability to unit test and transparently persist X10 programs for both Java and C++ backends, the X10 compilation targets.

The remainder of this paper is structured as follows. Section 2 defines the problem and sketches our solution. Section 3 details our design and implementation. Section 4 presents our case studies. Section 5 compares this work to the existing state of the art, and Section 6 concludes.

2. Problem Definition and Solution Overview

The programming model for implementing NFCs is becoming increasingly declarative. To add persistence, security, or testing to an application, programmers rarely write code in a mainstream programming language. Instead, programmers declare metadata such as XML files, Java 5 annotations, or C/C++ pragmas. Such a metadata declaration configures a standardized NFC implementation, provided as a library or a framework. 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 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. 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 addresses the problem described above. For this example, our approach 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 1.

This figure depicts the X10 class `FmmModel`[1] 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.

[1] `http://squirrel.anu.edu.au/hg/public/x10-apps/file/909f49fd95de/apps/fmm`

```
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 1. An X10 class to be compiled to Java and C++.

```
Metadata PersistentModelClasses<Package p>
  Class c in p
  Where (public class *Model)
    c += @class
    @class.name = c.name
    @class.table = 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 2. Metadata to persist the X10 class in Figure 1.

To that end, the programmer writes a metadata specification listed in Figure 2. We use Pattern-Based Structural Expressions (PBSE), a new metadata format we introduced recently [24] to improve the reusability, conciseness, and maintainability of metadata programming. PBSE leverages the correspondences between program constructs and metadata and uses queries on program structures to express how metadata should be applied. In Figure 2, the PBSE 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 a 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. 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,[2] an open source ORM system for C++ [3]. The **`pragma`** declarations specify how ODB should persist the fields of the C++ class emitted by the X10 compiler for the C++ backend.

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.

Our approach is intended to support the implementation of major NFCs that include security, transactions, and testing. Although we demonstrate our approach on the domain of unit testing and transparent persistence, our approach is general because of how NFCs are commonly implemented in modern languages. In particular, declarative approaches are common, with metadata being used as the preferred expression medium. Programmers use metadata, such as XML files, Java 5 annotations, C/C++ pragmas, macros, or C# attributes, to express how NFCs should be implemented in their programs.

For example, a C# security framework can provide special attributes for the programmer to restrict access to methods and fields. Once the programmer annotates the program, the framework will furnish the specified security functionality, thus implementing this NFC. If an emerging language compiled to C# needs to implement security, the methods and fields of the emerging language can be marked with the required access restrictions using any available metadata format. The resulting metadata specification in any format can then be translated to C# attributes that work with the C# code emitted by the source-to-source compiler.

3. Design and Implementation

In the following discussion, we first outline our requirements and design space considered, and then detail our implementation, including PBSE enhancements, metadata translation, and NFC API generation.

3.1 Design Objectives and General Approach

When designing our approach, we aimed at (1) providing a declarative programming interface, (2) maintaining generality, and (3) not imposing an unreasonable performance overhead. Specifically, our approach is designed to support those existing implementations of NFCs that expose a high level, declarative programming interface. Expressing major NFCs—including persistence, transactions, security, and testing—through metadata has become an industry practice.

[2] Surprisingly, ODB is not an acronym.

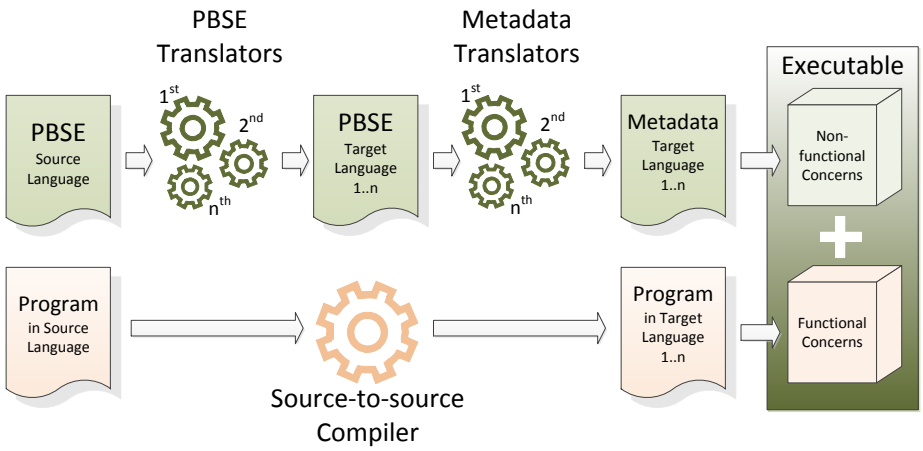

Figure 3. Generating Target Sources and Metadata formats.

Thus, our programming interface design goal was to enable the programmer to interface with our tool chain through declarative metadata specifications. We aimed at making our approach general with respect to both the NFCs it supports and the kinds of languages to which it applies. Finally, our approach should not impose an undue performance overhead on the target programs.

To support these design goals, we had to choose an appropriate metadata format that can be translated to the required metadata representations used by the existing NFC implementations in mainstream languages. To that end, we chose PBSE to support our goal of generality. PBSE is external to the source code and can work with any source languages, even if they do not provide built-in metadata constructs such as X10 annotations [18]. To fulfill this goal we could also have used XML files, but we found that XML is not a suitable format as a programmer written medium. PBSE provides conciseness, reusability, and maintainability advantages [24]. By capturing the naming correspondences between programming constructs and their metadata, PBSE is more expressive than mainstream metadata formats.

Because our approach hinges on the ability to effectively translate PBSE to existing metadata formats used with existing mainstream languages (e.g., annotations, pragmas, XML deployment descriptors, macros, etc.), we considered several choices with respect to designing our metadata translation infrastructure. Some emerging languages provide sophisticated facilities for systematically extending the core compiler. For example, the X10 compiler can be extended through plug-ins, but not all the emerging languages can be similarly extended. Striving for generality, we made our metadata translation infrastructure external to the source-to-source compiler, and ensured that the translation strategies are adaptable, customizable, and configurable.

In terms of the programming model, 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 Domain-Specific Language (DSL) that is derived from PBSE. The resulting PBSE mapping specification parameterizes a generator that synthesizes a PBSE cross-translator (Section 3.3). Finally, the emerging language programmers only need to declare PBSE to add any NFC implementation to their programs.

To increase flexibility without jeopardizing performance, we also automatically generate a special target language API for each supported NFC implementation rather than provide a pre-defined library. Automatically generating the API also makes it possible to introduce workarounds whenever an NFC implementation cannot be straightforwardly added to the target language programs. For example, Scala functional lists and maps translate to Java classes that cannot be directly persisted using mainstream Java persistence frameworks interfaces. They are translated to Java classes that do not implement the `java.util.List` and `java.util.Map`. Our code generator synthesizes mirror data structures compatible with Java persistence and copies the data back and forth during the saving and restoring operations.

3.2 Implementation Details

Figure 3 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 simultane-

ously. 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)[15] requires that the programmer use Java 5 annotations. Our approach can translate a PBSE 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.

3.3 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 4 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. If it were part of the language, such as in the case of Java 5 annotations, the program's source-to-source compiler would have to compile the metadata as well. If the format between the source and target metadata is not going to change (e.g., if it were an XML file for the source language program, it will also be an XML file for the resulting target language program), then the metadata's compiler must mirror the program's source-to source compiler transformations for the program's constructs tagged with metadata.

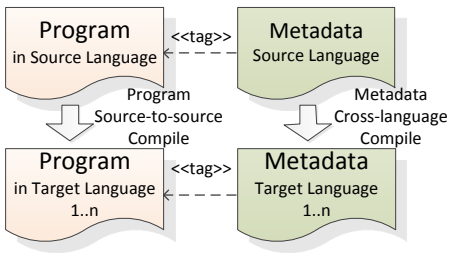

Figure 4. Translating Metadata formats.

In our approach, PBSE specifications can be translated across languages. In particular, 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.

Metadata translation framework Since metadata translation is the cornerstone of our approach, one of the key design goals we pursued was 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.

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. The meta-metadata in Figure 5 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.

```
1  MetaMetadata PBSEJavaToXML<PBSE pbse>
2    Where (Class c in pbse)
3      @Table -> "<class/>"
4      @Table.name -> "<class name=" + c.name + "/>"
5      @Table.class -> "<class table=" + c.table + "/>"
6    Where (Field f in pbse)
7      @Field -> "<field/>"
8      @Field.name -> "<field name=" + f.name + "/>"
9      @Column -> "<column/>"
10     @Column.name -> "<column name==" + f.column + "/>"
```

Figure 5. Meta-metadata for translating PBSE for Java to XML used by the JDO system.

Generative visitors Based on the meta-metadata specification in Figure 5, our code generator synthesizes a visitor class in shown Figure 6. 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

```
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){ /*..*/ }
  // other visit methods go here.
}}
```

Figure 6. A generated visitor.

classes such as `PBSEElementClass` and `PBSEElementField`. It also uses a utility class `JavaToXML` that encapsulates low-level translation functionality. The XML in Figure 7 was produced by one of the generated visitors.

Figure 8 presents a UML diagram of the visitors used in the examples discussed throughout the paper. 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.

```
<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 7. Translated XML metadata for the JDO system.

3.4 Discussion

Our approach leverages the prevalence of declarative abstractions for expressing NFCs in modern enterprise applications. In particular, the programmer expresses these concerns by declaring metadata. The expressed functionality is provided by libraries and frameworks, which heavily rely on code generation and transformation both at source or bytecode levels. For example, a specialized compiler or a bytecode enhancer can add persistence to an application as specified by a metadata declaration. Due to their conciseness and simplicity, declarative specifications are particularly amenable to automatic transformation, a property exploited by our approach.

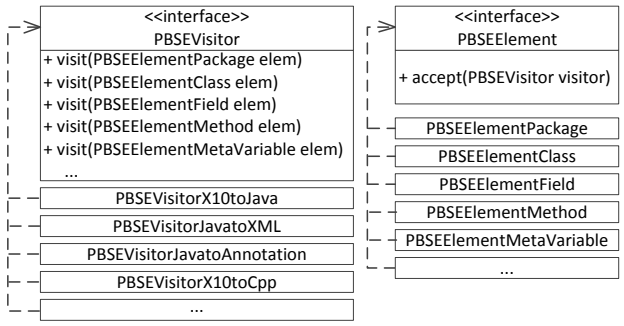

Figure 8. PBSE visitors translating metadata format.

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 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 certain core functionalities, metadata translation can supplement or, in some instances, replace compilation.

4. Case Studies

To validate our approach, we applied it to reuse four NFC implementations across two domains and two languages. We reused the JUnit and CppUnit testing frameworks, thereby adding unit testing capabilities to X10. We also reused Java Data Objects (JDO) and ODB, 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.

4.1 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.

Although testing has not been explicitly identified as an NFC in the literature, unit testing is indeed an NFC. Unit tests help ensure that a program does what it is expected to do, but they do not affect the program's core functionality. Adding unit testing to a program does not change the program's semantics. Furthermore, unit testing frameworks heavily rely on metadata used by the programmer to declare how a framework should run unit tests.

Consider the X10 class `Integrate` in Figure 9 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.[3] An area is computed by integrating its partial parts. For example, when computing the area with the start of a and the end of b, $\int_a^b f(x)dx$ computes partial results through integration. The application then sums up the partial integration results—$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{b-a}{2}x + \frac{b+a}{2})dx$.

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

Figure 9. An X10 Integrate class to be unit tested.

Gaussian quadrature is non-trivial to implement correctly, but this implementation is even more complex as it involves 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 [8]). The programmer should be able to write unit tests in an X10 program, and depending on the compilation target, compile these tests to be run by JUnit or CppUnit.

[3] `http://x10.svn.sourceforge.net/viewvc/x10/benchmarks/trunk/microbenchmarks/Integrate/`

Our approach makes it possible to reuse the implementations of this NFC. 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` in Figure 9 is shown in Figure 10.

```
1  public class IntegrateTest {
2    var parm : double;
3    var expt : double;
4    var integrate : Integrate;
5
6    def init() {integrate = new Integrate();}
7
8    def finish() {integrate = null;}
9
10   def this(parm : double, expt : double) {
11     this.parm = parm;
12     this.expt = expt;
13   }
14
15   public def testComputeArea() {
16     val result = integrate.computeArea(0, this.parm);
17     TUnit.assertEquals(this.expt, result);
18   }
19
20   public static def data() {
21     val parm = new Array[double](0..1*0..2);
22     parm(0, 0) = 2;
23     parm(0, 1) = 6.000000262757339;
24     parm(1, 0) = 4;
25     parm(1, 1) = 72.000000629253464;
26     parm(2, 0) = 6;
27     parm(2, 1) = 342.000001284044629;
28     return parm;
29 }}
```

Figure 10. The unit testing class for the X10 Integrate class.

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 multi-dimensional 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 11, the programmer also has to declare a simple metadata specification shown in Figure 12. This specification establishes a coding convention as the one used in class `IntegrateTest`. The main advantage of PBSE as compared to annotations is that 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. A key advantage of our approach is that it addresses the incongruity of features in different NFC implementations through code generation. 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 11). In addition, JUnit requires that the method pro-

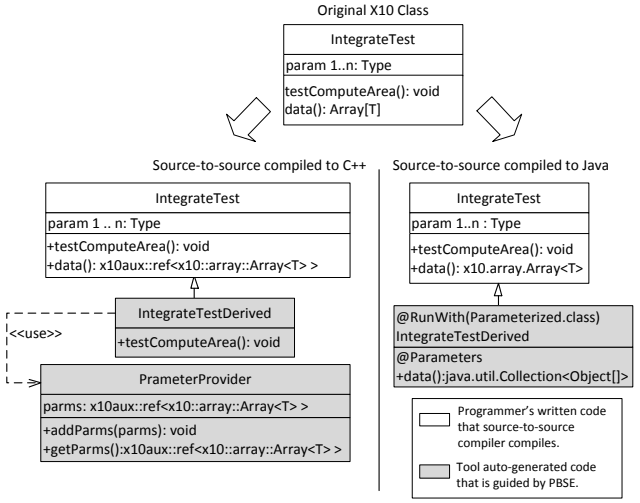

Figure 11. The class diagram for unit testing X10 programs with JUnit and CppUnit.

```
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 12. The PBSE for unit testing the X10 program.

viding the parameters for unit test instantiations return `java.util.Collection` (the right part of Figure 11). Unfortunately, the X10 method `data` is compiled to the Java method returning `x10.array.Array`, which does not extend this Java interface.

To overcome these limitations, our code generator synthesizes special classes that subclass the Java and C++ unit test classes generated by the X10 compiler. These methods provide the required functionality by leveraging the Adaptor design pattern. To ensure that the `data` methods return 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 13).

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

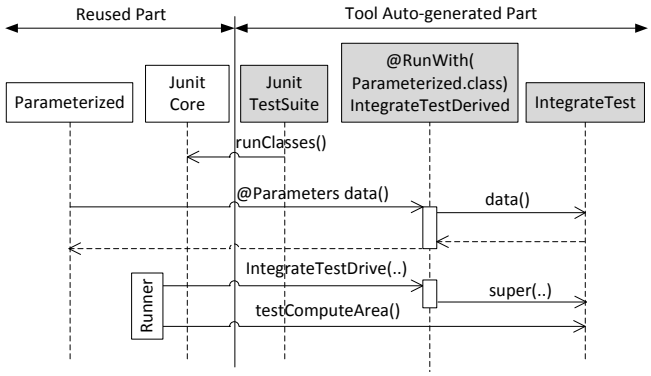

Figure 13. The sequence diagram for unit testing X10 programs with JUnit.

used to define a DSL for expressing functionality at a higher abstraction level.

The macros in Figure 14 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.[4] 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 15).

```
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 14. Extended macros based on CppUnit.

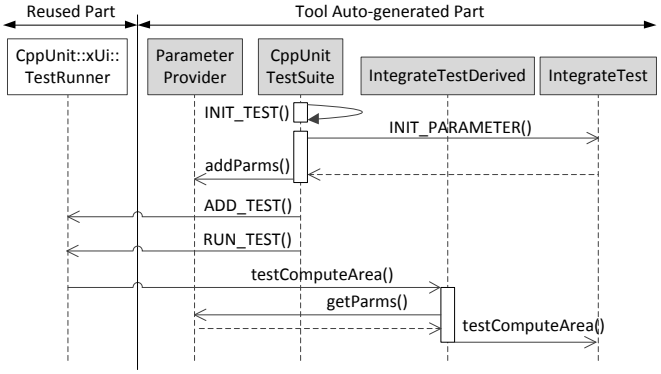

Figure 15. The sequence diagram for unit testing X10 programs with CppUnit.

Standard implementations of NFCs in richer languages expectedly provide more features and capabilities. In the

[4] These macros are regenerated from scratch for every PBSE translation.

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.

4.2 Transparently Persisting X10 Programs

Next we describe how we applied our approach to enhance X10 programs with transparent persistence capabilities. Both compilation targets of X10—Java and C++—use ORM engines to implement transparent persistence. Our approach makes it possible to reuse these implementations, thereby making X10 programs transparently persistent.

Figure 16 shows an X10 class `Fmm3d` that implements the Fast Multipole Method for electrostatic calculations with analytic expansions. The implementation is real and current: it follows the strategy outlined by White and Head-Gordon [25] which was recently enhanced by Lashuk et al. [14]. The `getDirectEnergy` method sums the value of direct energy—`directEnergy`—on line 4 for all pairs of atoms. This operation requires only that atoms be already assigned to boxes, and can be executed in parallel with the other steps of the algorithm.

```
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 +=
13           atom1.charge * atom2Packed.charge /
14           atom1.centre.distance(atom2Packed.centre);
15   }}}
16   model.setModelId(id(box1.x,box1.y,box1.z));
17   model.setEnergy(thisPlaceEnergy);
18   // other setter methods go here.
19   TP.setFmmModelObj(model);
20 }
21 offer thisPlaceEnergy;
22 }};
23 return directEnergy;
24 }}
```

Figure 16. A persisting class `Fmm3d` (simplified version) for the X10 `FmmModel` class in Figure 1.

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 2). 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)`.
Therefore, our approach shields the programmer from the idiosyncrasies of platform-specific 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/C++ pragmas. Nevertheless, our approach was able to seamlessly support these disparate metadata formats. Furthermore, the metadata specifications for both Java and C++ backends were automatically generated from the same PBSE X10 specification.

Figure 17 depicts a segment of the generated JDO XML deployment descriptor. To generate this deployment descriptor, our approach uses the PBSE depicted in Figure 18. Parameterized with this descriptor, the JDO runtime can transparently persist the specified X10 fields when the program is compiled to Java. Figure 19 depicts a segment of the generated ODB pragma definitions. To generate these pragmas, our approach uses the PBSE depicted in Figure 20. 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 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.

Because our approach uses template-based code generation, expert programmers can easily customize how the API is generated. The API generators take as input a PBSE specification and a code template. Our code templates provide several keywords that make it possible to flexibly parameterize the generator. For example, the database transaction API template in Figures 21 is parameterized with `$[Class.name]` and `$[Class.field.type]`.

The template features the `$iterator[..]` and `$parm` constructs that express how the same code can be generated for each collection element. These parameters are then matched with the specific program constructs tagged by a PBSE specification. Even though the API is generated in X10, its translated versions interoperate with the C++ and Java backends.

```
<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 17. 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 18. 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 19. 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 20. PBSE for transparent persistence in C++.

```
PersistenceManager pm = getPersistenceManager();
$1[Class.name] pobj = getObj(pm,$1[Class.name],$parm);
Transaction tx = pm.currentTransaction();;
tx.begin();
if (pobj == null) {
  pobj = new $1[Class.name]($iterator
              [$parm.$2[Class.field.type]]);
}
pm.makePersistent(pobj);
tx.commit();
```

```
auto_ptr < database >
     db (create_database (argc, argv));
$1[Class.name]* pobj = $parm._val;
transaction t(db->begin());
if (checkNull(pobj)) {
  db->persist(*pobj);
}
t.commit();
```

Figure 21. The code template for generating database transaction API for the Java (top) and C++ (bottom) backend.

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 for recovering from failure. In case of a crash to avoid restarting from the beginning, 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.

In our experiments, we added checkpointing capabilities to two X10 third-party applications: (1) *Fmm3d*, Fast Multipole Method for electrostatic calculations and (2) *SSCA1*,[5] the Smith-Waterman DNA sequence alignment algorithm.[23] SSCA1 computes the highest similarity scores by comparing in parallel an unknown sequence against a collection of known sequences. As discussed above, Fmm3d is Fast Multipole Method for distributed electrostatic calculations in a cubic simulation space centered at the origin.

In this benchmark, we measured the differences between the original and checkpointing-enabled versions of the applications, in terms of their respective total execution time. We also verified that the added checkpointing functionality does not negatively affect program scalability. Specifically, for both applications, we measured the total execution time of the original applications as well as their checkpointing-enabled versions, with the number of checkpoints increasing from 4 to 20 in the increments of 4.

[5] http://x10.svn.sourceforge.net/viewvc/x10/benchmarks/trunk/SSCA1/src-x10/ssca1/

The measurements were performed on Linux version 2.6.32-30, 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. As Figure 22 demonstrates, the checkpointing functionality implemented via transparent persistence neither incurs significant performance overhead nor hinders scalability. The incurred overhead remains constant for both Java and C++ backends.

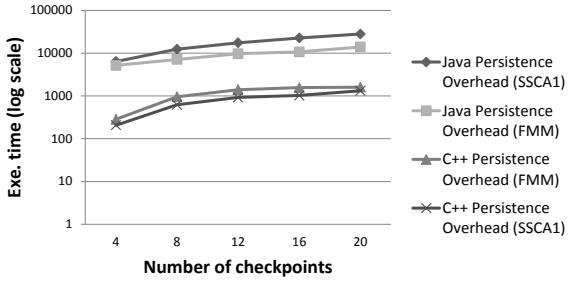

Figure 22. Persistence overhead for both backends.

5. Related Work

Our approach to reusing NFCs across languages is rooted in metadata translation, expressing NFCs via AOP, and code generation—an extensive body of related work. Thus, next we discuss only the closely related state of the art.

Metadata Translation Similarly to our approach, several prior approaches also leverage metadata translation, albeit not across languages. Godby et al. [6] 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 [17] 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. [21] transform across different metadata formats to achieve knowledge representation compatibility in different domains by means of domain knowledge. Hernández et al. [9] translate their custom metadata specifications for database mapping and queries.

Popa et al. [20] 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 cross-language metadata and provides meta-metadata to encode the translation rules. The objective of our approach is to bring the power of metadata translation to emerging source-to-source compiled languages, enabling the programmer to reuse complex NFC implementations declaratively.

Reusing Non-Functional Concerns with AOP Aspect-oriented Programming [12] is the foremost programming discipline for implementing NFCs. It has been debated which NFCs can be treated separately [13]. However, our approach reuses only those NFCs that have already been expressed separately in target languages. 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 [1] and JBoss AOP [10], 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 automatic code generation. The benefits of this technique are well-known in different domains.

Milosavljević et al. [16] 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 [7] 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 [4].

Devadithya et al. [5] 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 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 source-to-source compilation platforms.

6. Conclusions

In this paper, we have presented a novel approach to reusing NFC implementations across languages. Our approach en-

ables emerging language programmers to take advantage of such implementations in the target languages of a source-to-source compilation process. As a specific application of our approach, we added unit testing and transparent persistence to X10 programs, thereby reusing four existing, mainstream, NFCs implementations in Java and C++.

This paper contributes an approach to reusing NFCs implemented in a mainstream language from an emerging language program, when the emerging language is source-to-source compiled to the mainstream one; automated cross-language metadata translation—a novel approach to translating metadata alongside compiling the source language; meta-metadata that declaratively specify mappings between metadata formats; and the ability to unit test and transparently persist X10 programs for both Java and C++ backends.

The ongoing quest to bridge programmer imagination and computing capabilities motivates the continuous emergence of new programming languages. When an emerging language is source-to-source compiled to a mainstream one, the NFC implementations of the mainstream language remain inaccessible to the emerging language programmers. The presented novel approach reuses NFCs in mainstream languages by automatically translating metadata alongside compiling the source language. By eliminating the need to reimplement NFCs in emerging languages, our approach saves development effort.

Acknowledgments This research was sponsored through an IBM X10 Innovation Award. The IBM Research X10 team patiently answered our questions about the X10 language, compiler, and benchmarks. Fruitful discussions with our IBM liaison, Igor Peshansky, helped crystallize many of this paper's ideas. Boris Kolpackov helpfully guided us through our experiences with ODB.

References

[1] AspectJ Team. The AspectJ 5 development kit developer's notebook. `http://eclipse.org/aspectj/doc/next/adk15notebook/`.

[2] Charles Nutter, Thomas Enebo, Ola Bini and Nick Sieger. JRuby. `http://www.jruby.org/`.

[3] CodeSynthesis. ODB: C++ Object-Relational Mapping. `http://www.codesynthesis.com/products/odb/`.

[4] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Softw. Pract. Exper.*, 2004.

[5] T. Devadithya, K. Chiu, and W. Lu. C++ reflection for high performance problem solving environments. In *Proceedings of the spring simulation multiconference*, 2007.

[6] C. J. Godby, D. Smith, and E. Childress. Two paths to interoperable metadata. In *DCMI: Proceedings of the international conference on Dublin Core and metadata applications*, 2003.

[7] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[8] P. Hamill. *Unit test frameworks*. O'Reilly, first edition, 2004.

[9] M. A. Hernández, P. Papotti, and W. C. Tan. Data exchange with data-metadata translations. *Proc. VLDB Endow.*, 2008.

[10] JBoss. JBoss AOP. `http://www.jboss.org/jbossaop/`.

[11] Jython Project. Jython: Python for the Java Platform. `http://www.jython.org/`.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwing. Aspect-oriented programming. In *ECOOP: Proceedings of the 11th European Conference on Object-Oriented Programming*, 1997.

[13] J. Kienzle and R. Guerraoui. AOP: Does It Make Sense? The Case of Concurrency and Failures. In *ECOOP: Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002.

[14] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and 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.

[15] Maven. Security Annotation Framework. `http://safr.sourceforge.net/`.

[16] B. Milosavljević, M. Vidaković, and Z. Konjović. Automatic code generation for database-oriented web applications. In *Proceedings of the second workshop on Intermediate representation engineering for virtual machines*, 2002.

[17] K. Morik and M. Scholz. The miningmart approach to knowledge discovery in databases. In *Intelligent Technologies for Information Analysis*, 2003.

[18] N. Nystrom and V. Saraswat. An annotation and compiler plugin system for X10. Technical report, IBM TJ Watson Research Center, 2007.

[19] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, EPFL.

[20] L. Popa, Y. Velegrakis, M. A. Hernández, R. J. Miller, and R. Fagin. Translating web data. In *VLDB: Proceedings of the international conference on Very Large Data Bases*, 2002.

[21] T. Ruotsalo and E. Hyvönen. An event-based approach for semantic metadata interoperability. In *Proceedings of the 6th international semantic web conference*, 2007.

[22] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specication Version 2.1. Technical report, IBM TJ Watson Research Center, 2011.

[23] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Bwl*, 1981.

[24] E. Tilevich and M. Song. Reusable enterprise metadata with pattern-based structural expressions. In *AOSD: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, 2010.

[25] C. A. White and M. Head-Gordon. Derivation and efficient implementation of the fast multipole method. *The Journal of Chemical Physics*, 1994.