Supporting Effective Reuse and Safe Evolution in Metadata-Driven Software Development

Myoungkyu Song

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Eli Tilevich (Chair)
Stephen H. Edwards
Dennis Kafura
Barbara G. Ryder
Kwangkeun Yi

April 5, 2013
Blacksburg, Virginia

Keywords: declarative programming, metadata, domain-specific language, enhancement, program transformation, bytecode engineering, debugging, source-to-source translation, frameworks, transparent persistence, unit testing, refactoring, invariants

Supporting Effective Reuse and Safe Evolution in Metadata-Driven Software Development

Myoungkyu Song

ABSTRACT

In recent years, *metadata-driven software development* has gained prominence. In this implementation model, various application concerns are provided as third-party frameworks and libraries that the programmer configures through metadata, such as XML configuration files or Java annotations. Metadata-driven software development is a special case of *declarative programming*: metadata serves as a domain-specific language that the programmer uses to declare various concerns, whose implementation is provided by an elaborate ecosystem of libraries and frameworks that serve as pre-defined application building blocks. 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. Metadata-driven software development has been particularly embraced in enterprise computing as a means of providing standardized solutions to common application scenarios.

Despite the conciseness and simplicity benefits of metadata-driven software development, this implementation model introduces a unique set of reuse and evolution challenges. In particular, metadata is not reusable across application modules, and program evolution causes unsafe discrepancies between the main source code and its corresponding metadata. The research described in this dissertation addresses five fundamental problems of metadata-driven software development: (1) bytecode enhancements that transparently introduce concerns hinder program understanding and debugging; (2) mainstream enterprise metadata formats are hard to understand, evolve, and reuse; (3) concerns declared via metadata cannot be reused when source-to-source compiling emerging languages to mainstream ones; (4) metadata correctness cannot be automatically ensured as application source code is being refactored and enhanced; and (5) lacking built-in metadata, JavaScript programs can be enhanced with additional concerns only through manual source code changes.

The research described in this dissertation leverages domain-specific languages and automated code generation to enable effective reuse and safe evolution in metadata-driven software development. The specific innovations that address the problems outlined above are as follows: (1) a domain-specific language (DSL) describing bytecode enhancement that facilitates the understanding and debugging of additional concerns; (2) a novel metadata format expressed as a DSL that is easier to author, understand, reuse, and maintain than existing metadata formats; (3) automated metadata translation that enables effective reuse of target language additional concerns from source-to-source compiled source language programs; (4) metadata invariants—a new abstraction for expressing and verifying metadata coding convention; and (5) a new approach to declaratively enhancing JavaScript programs with additional concerns.

This dissertation is based on research papers that appeared at OOPSLA'09 [128], AOSD'10 [148], AOSD'12 [131], and ICSE'12 [130], and software demos presented at OOPSLA'09 [127] and ICSE'12 [129].

ACKNOWLEDGEMENTS

I wish to express my gratitude to my advisor and committee members who supported me during my doctoral research.

Firstly, I would like to express my deepest appreciation for my advisor and research mentor, Eli Tilevich, for his valuable support, prompt guidance, and effective encouragement throughout my graduate program. He has supported me on every aspects of my Ph.D. dissertation research, in particular with emphasis on guiding me to the true research process. He always made himself available whenever I needed his advice, selflessly introducing substantial amount of time guiding this research, regardless to weekends and holidays.

He has his special mentoring skill that he can listen not only what students are saying, but also what students are not saying. His guidance encompassed such diverse perspectives. I have learned how research ideas should be selected and developed to publish top-tier conferences. And, his practical and thoughtful mentoring has also led me to participate in a couple of research proposals.

I also thank my committee members, Steve H. Edwards, Dennis Kafura, Barbar Ryder, and Kwangkeun Yi, for carefully reading my dissertation and presenting constructive feedbacks and valuable insight that made it possible for me to improve the overall quality of the dissertation research.

Steve H. Edwards provided the essential assistance in improving this dissertation manuscript, especially on the chapters, Introduction, Case Studies and Conclusion. Sharing with me his insightful comments about this research helped me clarify the general contributions of this work.

My special thanks go to Barbara G. Ryder, our department head, who took time from her busy schedule to provide me with constructive help in enhancing my research papers and addressing future work directions being applicable to my dissertation.

My graduate journey would have not been possible without warmest care from other faculty members, and staffs, as well as my family, colleagues, and friends.

I would like to thank all my lab mates and VT CS friends, as well as all my Korean graduate friends, for their friendship and countless help. I improved my all practice talks by means of the comments raised by them, with whom I also enjoyed brainstorming research ideas.

I would not have gotten that far without the support of my family. I thank my lovely wife, Hyeonsu Park. She made my life much more dynamic, which could turn to encouragement during the past few years. I thank my parents' constant support and my sisters' help, while patiently waiting for me to complete my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The reality of modern software development in multiple application domains requires delivering software in the presence of tight deadlines, limited budgets, and scarce programming talent. State-of-the-art software engineering techniques and tools (e.g., programming conventions, reusable libraries, application frameworks, code generators, bug finding tools, etc.) are continuously created and perfected to address these challenges. One such technique is *the declarative programming model*, in which programmers declare the required functionality at a high level of abstraction, with the actual implementation provided by automated tools such as libraries, program transformation frameworks, and code generators.

A key benefit of the declarative programming model is that it enables a clear separation of concerns, a design and implementation principle that promotes treating individual pieces of application functionality independently from each other. In industrial software development, various non-functional requirements (e.g., persistence, security, transactions, etc.) were shown to lend themselves well to be implemented declaratively. Other common software engineering tasks such as unit testing are being implemented increasingly declaratively as well. Thus, the declarative programming model has become an integral part of modern industrial software development.

In modern software development practices, the declarative programming model is commonly expressed by means of metadata, such as XML, Java 5 annotations, C# attributes, and C/C++ pragmas. Metadata declaratively identifies the program constructs (e.g., classes, methods, fields, etc.) as the program points at which additional concerns are to be introduced. As an example, persistence frameworks automatically persist the fields of a class annotated with the `@Column` Java Persistence Architecture (JPA) [138] annotation. Programmers only need to declare which object fields correspond to relational database columns, and persistence frameworks supply all the required functionality to render the annotated fields persistent.

In essence, metadata codifies a domain-specific language (a language specially designed for solving problems in a given domain) for expressing various concerns declaratively. All in all, metadata constitutes an important building block of modern software applications. Software developers remain oblivious as to how the declarative metadata directives are translated into specific concern implementations, but instead spend their time and efforts on writing and maintaining metadata. De-

spite being the foundation of the declarative programming model, metadata incurs unique software maintenance and evolution burdens that cannot be addressed with the existing software maintenance techniques, approaches, and tools.

One such problem is that mainstream metadata formats do not lend themselves easily to systematic reuse. For example, in J2EE[143], a set of standards for developing enterprise applications in Java, XML deployment descriptors specify how individual Java classes interface with frameworks that implement various concerns. Especially, programmers write XML files for each program class that uses any framework functionality. When using Java annotations, programmers must not only annotate each program separately, but they also have to 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 need to repeat declarations not only creates surplus work for the programmer, but also increases the probability of introducing errors.

Due to its domain-specific nature, mainstream metadata cannot be handled by source-to-source compilers, commonly used for implementing emerging programming languages. Source-to-source compilers are domain-independent, as they translate languages based on the general semantic of their program constructs. As a result, source language programs cannot reuse the functionalities configured by means of metadata in the target language. Consequently, emerging language programmers must reimplement these functionalities from scratch, an undertaking that requires porting numerous complex libraries and frameworks to the emerging language.

Additionally, there is the problem of ensuring that metadata maintains its integrity and correctness during software evolution. In particular, when a program evolves, its main source code and metadata may become inconsistent, introducing insidious, difficult-to-catch bugs, compromising software utility and reliability. Existing, mainstream methodologies, techniques, and tools for managing software evolution have not been designed with metadata in mind.

Finally, as web applications now constitute an integral part of the modern computing infrastructure, JavaScript programs keep growing in size and complexity. Multiple JavaScript libraries have been introduced for implementing various application concerns. To use these libraries, programmer modify the original JavaScript source code by hand. Adding different concerns to the same JavaScript code base by hand creates divergent program versions that must be maintained separately. Thus, the declarative programming model, in which the same codebase can be declared to include different concerns, can potentially benefit JavaScript programs as well. However, lacking built-in metadata, JavaScript is poorly fit for declarative programming.

1.1 Background

In this section, we introduce the main concepts and technologies used in the research described in this dissertation: enterprise applications, software reuse, bytecode enhancement, domain-specific languages, and automatic code generation.

Enterprise applications help enterprises improve the efficiency of their workflows. The spectrum of enterprises that has benefited from such applications ranges from small on-line stores to large, multinational corporations. In recent years, enterprise computing has embraced the practices of metadata-driven software development as shown in Figure 1.1.

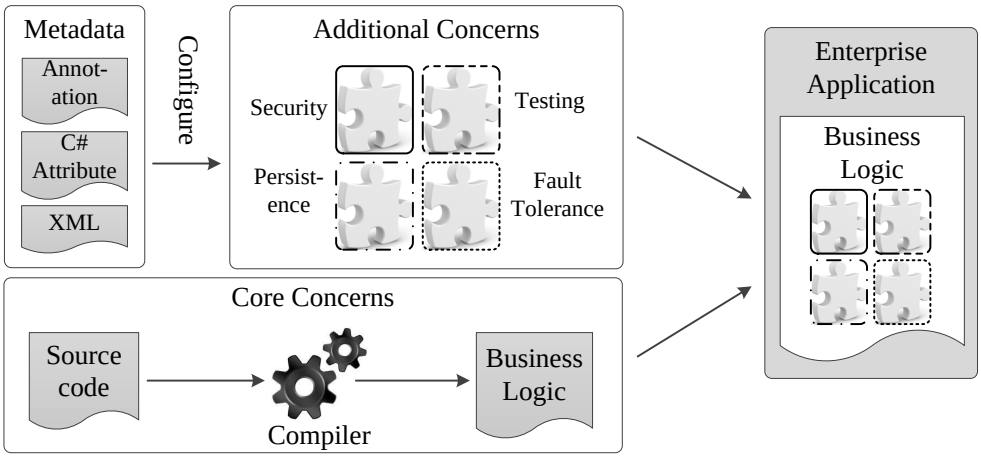

Figure 1.1: Metadata-driven software development.

Whenever striving to improve software quality or increase productivity, programmers commonly resort to reusing existing software, a practice known as software reuse [48]. 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 [120]. Reusing tested software pieces in a software product improves the product's overall quality [21]. 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 [113, 110, 112, 7], libraries [20, 54, 55], and specifications [156, 36, 26]. In this dissertation, we explore the issues of reusing

yet another artifact of modern software construction—metadata, which comes in many formats, such as XML, Java 5 annotations, etc.

The majority of enterprise applications are written in managed languages such as Java and C# that are compiled to intermediate representation such as bytecode. A technique called bytecode enhancement transparently modifies the application's bytecode to introduce additional concerns. Alternatively, special libraries can be configured through metadata and added to the compiled bytecode. This dissertation addresses the software maintenance problems that arise from the differences between the programmer written code and its enhanced version executed at runtime.

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 single domain, such as database programming—SQL[77], statistical computations—R[47], and text processing—LaTeX[78]. As compared to general purpose languages, a DSL can solve a domain problem more concisely with a reduced programming effort [15, 45]. DSLs have been successfully employed to solve software construction challenges in multiple domains. This dissertation explores how the expressive power of DSLs can be exploited to address the issues pertaining to reusing concern implementations in metadata-driven software development.

One of the most effective approaches to automating menial programming tasks is automatic code generation. A code generator takes a high level description as input and generates lower level code. That is, the input specification for generators is simpler and shorter than the generated code. Hence, code generation not only saves time and effort, but also avoids many programming errors and increases programmer productivity [124, 123]. As the complexity of computing systems continues to grow, software generators have the potential to provide elegant solutions that tame the complexity, enabling the design at a higher level of abstraction and the use of declarative approaches [124]. This dissertation explores how automated code generation can alleviate some of the problems pertaining to reuse and evolution in metadata-driven software development.

1.2 Research Agenda

As software development has become heavily dependent on the declarative model for implementing mission-critical functionality, the role of metadata has gained prominence. Indeed, programmers spend a substantial amount of their time and efforts writing, understanding, and evolving

metadata. As a result, systematic approach to metadata reuse and evolution can yield tangible software engineering benefits. This dissertation studies several key problems of metadata-driven software development, with an emphasis on effective metadata reuse and safe program evolution. This work examines why the existing state of the art in metadata-driven software development poorly supports reuse and is vulnerable to evolution-related issues; it then proposes novel solutions to these problems. In the following discussion, we summarize the research agenda of this dissertation by first presenting the main technical questions it addresses and then outlining the solutions. Section 1.3 provides additional details for the solutions.

- **How does metadata-driven software development affect software maintenance tasks, such as symbolic debugging?**
  - In metadata-driven software development, the framework functionality is commonly introduced directly to the bytecode of a program, a practice that results in runtime execution whose semantics differs from that expressed by the source code. As a result, the programmer cannot directly understand the program's behavior by browsing its source code, nor map its execution to the source code.
- **How does metadata-driven software development affect software reuse?**
  - Mainstream metadata is not reusable, not only across applications but even across different parts of the same program.
  - When source-to-source compiling an emerging language to a mainstream language, the concerns expressed by metadata in the target language programs cannot be reused by the emerging language programs.
- **How does metadata-driven software development affect the safety of common software evolution tasks, such as refactoring and enhancement?**
  - The main source code and their corresponding metadata interconnect so tightly that neither one can be safely evolved independently. Since metadata coding conventions are domain-specific, their violations cannot be discovered by existing compilers or bug finding tools.
- **Can the benefits of metadata-driven software development be made available in languages that lack built-in metadata?**

  - Introducing concerns to JavaScript programs requires modifying the base source code by hand, thus creating divergent versions of the codebase that must be maintained separately, thus increasing the maintenance burden. Declarative programming is inapplicable to JavaScript, which lacks built-in metadata.

The broader impact of this dissertation will help practicing software engineers to improve both their productivity (by enabling effective reuse of metadata) and software quality (by enabling safer evolution of software that relies on metadata). From the technical perspective, this research leverages the power of DSLs and automated code generation to address the aforementioned problems;

- **Facilitating Corrective Maintenance in Metadata-Driven Software Development**
  - A novel DSL expresses structural program enhancements—framework functionalities introduced at the binary level, thus improving the precision and utility of software tools such as symbolic debuggers and code browsers. The DSL serves as a foundation for a novel debugging architecture that enables symbolic debugging of programs whose intermediate code has been transparently enhanced. Another contribution is a special code browser that informs the programmer about how various program constructs will be enhanced at the bytecode level.
- **Enabling Effective Reuse in Metadata-Driven Software Development**
  - An alternative to mainstream metadata formats, Pattern-Based Structure Expressions (PBSE) offers several software engineering advantages compared to XML and Java annotations. This novel metadata format leverages the common patterns between the source code and its metadata exhibited by modern enterprise framework applications. By explicitly capturing and expressing these patterns, PBSE conveys metadata information declaratively and can be reused not only within the same application, but also across other applications that use the same enterprise frameworks.
  - A novel approach effectively reuses concerns expressed via metadata across languages by translating metadata, when emerging languages are source-to-source compiled to mainstream ones. This makes it possible for emerging language to take advantage of standardized, fine-tuned implementations of common concerns (e.g., persistence, security, transactions, and testing) in mainstreams languages.
- **Enabling Safer Software Evolution in Metadata-Driven Software Development**

  - A novel DSL expresses *Metadata Invariants*, which codify the metadata naming conventions that commonly occur between metadata and the source code it tags. The DSL can help identify metadata-related bugs and inconsistencies for the programmer to examine, especially during program evolution. Metadata invariants also can be automatically inferred by analyzing large codebases.

- **Availing Benefits of Metadata-Driven Software Development to Languages without Metadata**
  - A declarative approach to enhancing JavaScript programs applies the Java annotations infrastructure to JavaScript, without extending the JavaScript language; programmers declare how concerns should be added to a JavaScript program using Java annotations. Based on the annotations, a code generator synthesizes aspect code that adds the specified concerns.

The solutions outlined above are evaluated as follows:

1. The applicability of the enhancements-aware programming tools are evaluated by augmenting existing source-level programming tools—a code editor and a symbolic debugger—with an awareness of bytecode enhancements.

2. The reusability of PBSE is evaluated by expressing the transparent persistence functionality for a reference enterprise application and comparing the resulting implementation with those that use standard metadata.

3. The effectiveness of reusing additional concerns across languages is evaluated by adding concerns to third-party emerging language (X10) programs, by reusing implementations of these additional concerns in the source-to-source compilation target languages (C++ and Java).

4. The effectiveness of the metadata invariant inference algorithm is assessed with third-party, real-world enterprise applications that rely on metadata. The utility of metadata invariants checking is demonstrated by integrating this functionality as an IDE plug-in that displays the violated metadata invariants in the error window.

5. The effectiveness of declarative enhancement of JavaScript programs is evaluated based on the ability of this approach to introduce concerns declaratively thus eliminating the need—

the concern developer's effort, to manually modify the maintained version of the JavaScript source code.

1.3 Major Research Contributions

In the following discussion, we provide an overview of the major contributions made by this dissertation, which include an approach to understanding transparent bytecode transformations, reusable enterprise metadata format, an approach to reusing additional concerns across languages, an abstraction for metadata coding convention, and an approach to declaratively enhancing JavaScript programs with additional concerns.

1.3.1 Facilitating Corrective Maintenance in Metadata-Driven Software Development

Enhancing Source-Level Programming Tools with an Awareness of Transparent Program Transformations Programs written in managed languages are compiled to a platform-independent intermediate representation, such as Java bytecode. The relative high level of Java bytecode has engendered a widespread practice of changing the bytecode directly, without modifying the maintained version of the source code. This practice, called *bytecode engineering* or *enhancement*, has become indispensable in transparently introducing various concerns, including persistence, distribution, and security. For example, transparent persistence architectures help avoid the entanglement of business and persistence logic in the source code by changing the bytecode directly to synchronize objects with stable storage. With functionality added directly at the bytecode level, the source code reflects only partial semantics of the program. Specifically, the programmer can neither ascertain the program's runtime behavior by browsing its source code, nor map the runtime behavior back to the original source code. This research presents an approach that improves the utility of source-level programming tools by providing enhancement specifications written in the Structural Enhancement Rule (SER) language, a Domain-Specific Language (DSL). By interpreting the specifications, a source-level programming tool can gain an awareness of the bytecode enhancements and improve its precision and usability. We demonstrate the applicability of our approach by making a source code editor and a symbolic debugger enhancements-aware.

1.3.2 Enabling Effective Reuse in Metadata-Driven Software Development

Reusable Enterprise Metadata with Pattern-Based Structural Expressions An essential part of modern enterprise software development is metadata. Mainstream metadata formats, including XML deployment descriptors and Java 5 annotations, suffer from a number of limitations that complicate the development and maintenance of enterprise applications. Their key problem is that they make it impossible to *reuse* metadata specifications not only across different applications but even across smaller program constructs such as classes or methods. To provide better enterprise metadata, we present *pattern-based structural expressions (PBSE)*, a novel metadata representation that offers conciseness and maintainability advantages and is reusable. To apply PBSE to enterprise applications, we translate PBSE specifications to Java annotations, with annotating classes automatically as an intermediate build step. We demonstrate the advantages of the new metadata format by assessing its conciseness and reusability, as compared to XML and annotations, in the task of expressing metadata of J2EE reference applications and a mid-size, commercial, enterprise application.

Reusing Additional Concerns Across Languages Emerging languages are often source-to-source compiled to mainstream ones, which offer standardized, fine-tuned implementations of additional concerns—including persistence, security, transactions, and testing. Because these additional concerns are specified through metadata such as XML configuration files, compiling an emerging language to a mainstream one does not include additional concern implementations. Unable to access the mainstream language's additional concern implementations, emerging language programmers waste development effort reimplementing additional concerns. In this research, we present a novel approach to reusing additional concern implementations across languages by automatically translating metadata. To add an additional concern to an emerging language program, the programmer declares metadata, which is then translated to reuse the specified additional concern implementation in the source-to-source compiled mainstream target language program. By automatically translating metadata, our approach eliminates the need to reimplement additional concerns in the emerging language. As a validation, we add unit testing and transparent persistence to X10 by reusing implementations of these additional concerns in Java and C++, the X10 backend compilation targets. The reused persistence additional concerns 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 additional concern implementations across languages, thus saving development effort.

1.3.3 Enabling Safer Evolution in Metadata-Driven Software Development

Metadata Invariants: Checking and Inferring Metadata Coding Conventions As the prevailing programming model of enterprise applications is becoming more declarative, programmers are spending an increasing amount of their time and efforts writing and maintaining metadata, such as XML or annotations. Although metadata is a cornerstone of modern software, automatic bug finding tools cannot ensure that metadata maintains its correctness during refactoring and enhancement. To address this shortcoming, this research presents *metadata invariants*, a new abstraction that codifies various naming and typing relationships between metadata and the main source code of a program. We reify this abstraction as a domain-specific language that we call the Metadata Invariants Language (MIL). We also introduce algorithms to infer likely metadata invariants and to apply them to check metadata correctness in the presence of program evolution. We demonstrate how metadata invariant checking can help ensure that metadata remains consistent and correct during program evolution; it finds metadata-related inconsistencies and recommends how they should be corrected. Similar to static bug finding tools, a metadata invariant checker identifies metadata-related bugs as a program is being refactored and enhanced. Because metadata is omnipresent in modern software applications, our approach can help ensure the overall consistency and correctness of software as it evolves.

1.3.4 Availing Benefits of Metadata-Driven Software Development to Languages without Metadata

Declarative Enhancement for JavaScript Programs Recent state-of-the-art approaches enhance JavaScript programs with concerns (e.g., persistence, security, transactions, etc.) by modifying the source code by hand to use special libraries. As a result, adding concerns to a JavaScript program creates divergent codebases that must be maintained separately. At the core of the problem is that JavaScript lacks metadata to express concerns declaratively. In this research, we present *TAE-JS* (**T**ransparent **A**utomated **E**nhancement for **J**ava**S**cript), a declarative approach to enhancing JavaScript programs that applies the Java annotations infrastructure to JavaScript, without extending the JavaScript language. An IDE combines JavaScript and Java during the development, but processes the languages separately. Programmers declare how concerns should be added to a JavaScript program using Java annotations. Based on the annotations, a code generator synthesizes aspect code that adds the specified concerns. Although these enhancements are implemented as

third-party libraries, our approach can transparently insert them into JavaScript programs given a declarative specification.

1.4 Broader Impacts

By enabling the declarative programming model, metadata is one of the cornerstones of modern software, but unfortunately it is not properly supported by automated approaches that can verify its integrity during program evolution. The intellectual merit of this dissertation is concerned with systematically extending the state-of-the-art in reuse and evolution with respect to metadata-driven software development.

The solutions presented in this dissertation are defined at a fundamental, conceptual level, making them applicable to a wide range of languages and applications. The utility of this research is verified empirically against realistic applications. As this research activities aims at successful support of evolution complexity in metadata-driven software development, practicing software developers can be equipped with powerful software methodologies, techniques, and tools that ensure an awareness of transparent program transformed specified by metadata, a reusable metadata format across components, applications, and languages, automated cross-language metadata translation, safe evolution of applications using metadata, and declarative enhancements of JavaScript programs based on the Java annotation infrastructure.

The expected benefits include (1) precision and utility of programming tools, as making the tools aware of transparent bytecode enhancement help programmers enjoy the benefits of transparent bytecode enhancement without suffering any of its disadvantages, (2) reducing a substantial amount of time and efforts implementing and maintaining metadata, as enterprise software development has become heavily dependent on the role of metadata, (3) eliminating the need to reimplement additional concerns in emerging languages, as automated translating metadata alongside compiling the source language makes it possible to reuse additional concerns in mainstream languages, (4) increasing programmer productivity, as the efforts expended on verifying metadata correctness during program evolution can be significantly reduced, and (5) cleanly separated concerns when enhancing JavaScript code with additional concerns, as the metadata infrastructure of a foreign programming language (Java) is embedded in a host language (JavaScript), without modifying the host language's syntax.

1.5 Structure of This Document

The rest of this dissertation is structured as follows. Chapter 2 compares this research with the existing state of the art. Chapters 3, 4, 5, 6, and 7 covers motivations, design, and implementation of SER, PBSE, the automated cross-language metadata translation, metadata invariants, and TAE-JS. Chapter 8 reflects on our experiences of implementing and evaluating this research through case studies. Chapter 9 presents concluding remarks, the contributions of this research, and discuses future work directions.

Chapter 2

Literature Review

The objective of this chapter is to put the research described in this dissertation into perspective by showing how it relates to existing work. First, we discuss directly related work as pertaining to each of the components explored by this dissertation. Then we identify how this work on metadata-driven software development utilizes approaches and techniques from different research areas. Finally, we outline how this work could benefit or influence various areas of research and practice.

2.1 Enhancing Source-Level Programming Tools with an Awareness of Transparent Program Transformations

Our approach to augmenting source-level programming tools with an awareness of intermediate code enhancements is related to several research areas including the debugging of transformed code, program transformation languages, and structural program differencing.

2.1.1 Debugging Transformed Code

Debugging Optimized Code. Modern compilers have powerful code optimization capabilities. Compilers optimize code via performance-improving transformations. However, because compilers transform an intermediate representation of a program, the relationship between such a transformed representation and the original source code of a program becomes obscured. Specifically, performance-improving transformations reorder and delete existing code as well as insert new code. Thus, debugging optimized code is hindered by the changes in data values and code location. A significant amount of research aims at the problem of debugging optimized code [57, 59, 16, 80, 43, 28, 49]. The proposed solutions enable source-level debuggers to display the information about an optimized program, as if the original (unoptimized) version of the code was being debugged.

The techniques for debugging optimized code deal with the challenges arising as a result of optimizing compilers transforming intermediate code to improve performance. While these transformations can be quite extensive, they are usually confined to method bodies and do not alter the debugged program's semantics. By contrast, structural enhancement aims at larger-scale program transformations that can add new classes and interfaces to a program. Thus, source level debugging of structurally enhanced programs requires different debugging techniques such as the presented symbolic undo.

Debugging for AOP. Aspect-Oriented Programming (AOP) [73] can be viewed as an enhancement technology, and several approaches aim at debugging support for AOP [38, 109, 66]. However, debugging aspect-oriented programs is different from debugging transparently enhanced programs. AOP provides a domain-specific language for programming enhancements such as AspectJ [71], and an AOP debugger traces the bytecode generated by AspectJ to its aspect source file. By contrast, transparent bytecode enhancements have no source code representation. Thus, the approaches to debugging AOP software cannot be applied to debugging transparently enhanced programs.

In addition, AspectJ as a language cannot express all the structural enhancement transformations. For example, it is impossible to express in AspectJ that the name of a program construct (e.g., class, field, or method) be changed. AspectJ does not provide facilities for removing a program construct, something that program enhancers need to do on a regular basis. For example, the split class enhancement moves fields from a class to another class, thus removing them from the original class. Finally, program enhancers often generate new classes and interfaces and reference them in the enhanced program. AspectJ is not designed for expressing how to generate a new class, whose structure is based on some existing program construct. Thus, we could not have used AspectJ instead of SER to represent structural enhancements.

2.1.2 Program Transformation Languages

SER is a domain-specific language for expressing how bytecode is enhanced structurally. Other domains also feature domain-specific languages for expressing program transformations.

JunGL [151], a domain-specific language for refactoring, combines functional and logic query language idioms. JunGL represents programs as graphs and manipulates refactorings via graph transformations. The language provides predicates to facilitate the querying of graph structures. JunGL uses demand-driven evaluation to prevent scripts from becoming prohibitively complex.

Sittampalam *et al.* [122] specify program transformations declaratively and generate executable

program transformers from specifications. The Prolog language is augmented with facilities for incremental evaluation of regular path queries. The augmented language is used to specify program transformations by expressing a program and its transformed counterpart. The transformations can be combined with a strategy script, based on Stratego [152], to specify the traversals of a program and the order of the transformations.

Whitfield and Soffa's code-improving transformation framework consists of a case tool and a specification language [157]. The specification language, called Gospel, declares program transformations; the case tool, called Genesis, generates a program transformer given a Gospel specification. A Gospel script consists of a declaration section, containing variables declarations, a precondition section, containing code pattern descriptions and control dependence conditions, and an action section, containing a set of transformation operations.

The Coccinelle [135] tool introduces the SmPL language for locating and automatically fixing bugs. SmPL programs specify how in response to some runtime condition, a program should be transformed to fix a bug and log the changes.

FSMLs [30] is a domain specific modeling language for describing the framework-provided knowledge, including framework instantiation, procedures for implementing interfaces, and proper usage of framework services. FSMLs bears similarity to our approach in its ability to provide a bi-directional mapping between framework features and their abstract representation.

JAVACOP [2] introduces a declarative rule language for expressing programmer-defined types, called *pluggable types*. The types are described as user-defined rules, which JAVACOP uses for transforming the abstract syntax tree of a program.

Compile-Time Reflection (CTR) [41] enables generative programming without the intricacies of the reflection API in the mainstream managed languages such as Java or C#. To that end, CTR extends C# with the ability to inspect and generate code using templates and pattern matching. CTR generates code at compile time, thus ensuring that the generated code is statically checked. The purpose of CTR is to express how source code should be generated, rather than how bytecode should be enhanced.

Because these program transformation languages were designed specifically for their respective domains, we could not have used any of them for our purposes. The main design goal of our SER language was to be able to express structural enhancements used by bytecode enhancers declaratively and to provide special constructs for domain-specific transformations such as adding getter/setter methods.

2.1.3 Program Differencing

SER scripts describe generalized structural program differences. Program differencing is an active research area and several new differencing algorithms have been proposed recently.

Previtali *et al.*'s technique [111] generates version differences of a class at bytecode level. Their algorithm produces the information on added, removed, or modified classes. Dmitriev incorporates program change history into a Java make utility that selectively recompiles dependent source files [34]. The JDIFF [4] algorithm identifies changes between two versions of an object-oriented program using an augmented representation of a control-flow graph. M. Kim *et al.* [75] infer generalized structural changes at or above the level of a method header, represented as first-order relational logic rules. These techniques could be leveraged to generate SER scripts automatically by generalizing the differences between the original and enhanced versions of multiple classes.

Our own Rosemari system [144] generalizes structural differences between two versions of a representative example. Such examples are usually supplied by framework vendors to guide the developers in upgrading their legacy applications from one framework version to another. Rosemari features a DSL for describing program transformations, but can be retargeted to present structural changes in SER instead.

2.1.4 Symbolic Execution

The idea of symbolic undo, used in implementing our debugger, is influenced by symbolic execution [76], which analyzes a program by executing it with symbolic inputs, but without actually running it. By analogy, our symbolic undo technique enables source level debugging of enhanced bytecode by mapping it back, via symbolic operations, to its original source code, also without affecting the program's execution.

2.2 Reusable Enterprise Metadata with Pattern-Based Structural Expressions

Both AspectJ 5 [145] and JBoss AOP [67] provide language support for introducing annotations. AspectJ 5 does so thorough the `declare annotation` construct. JBoss AOP provides two ways

to introduce annotations: using XML with the `annotation-introduction` tag and using a special meta-annotation. Both of these AOP languages provide wildcards to specify a set of program constructs to which annotations are introduced. Although these wildcards enable greater flexibility in expressing the set of annotated program constructs, the resulting pointcut expressions are not easily reusable. While one can express, for example, that a certain annotation be added to all the methods matching a certain name pattern, the resulting aspect program constructs are not parameterizable—one cannot reuse them with a different class. In addition, neither AspectJ 5 nor JBoss AOP make it possible to express annotation names and attributes as a function of the annotated program constructs. Overall, the pointcut expressions used in introducing annotations in AspectJ 5 and JBoss AOP do not capture the structural dependencies between the annotations and the annotated program constructs. As a result, neither of these AOP languages can express reusable enterprise framework metadata as concisely as can PBSE.

In addition, AspectJ 5 can match join points based on annotations by including pointcuts for all types of Java 5 annotations. The support of AspectJ for join point matching based on annotations may help manage some of the shortcomings of annotations as a metadata format. While AspectJ does not attempt to create a better metadata representation for enterprise frameworks, this was our intent behind creating PBSE.

PBSE is related to several research efforts whose objective is to validate the correctness of metadata. Eichberg et al. [40] 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. [99] 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 5 annotations or .NET custom attributes, are not sufficiently expressive. This is the problem that PBSE aims to address. PBSE encodes the structural dependencies between program constructs and their corresponding metadata. By encoding such correspondences explicitly, PBSE specifica-

tions are less likely to contain unexpected inconsistencies and bugs. Furthermore, the declarative nature of PBSE makes it easier to ascertain complex metadata invariants by examining a single PBSE specification.

Pattern-based reflective declaration [42, 64, 63] extends C# and Java to declare program constructs such as fields and methods as a static, pattern-based, reflective iteration over other classes. Pattern-based reflective declaration is a meta-programming technique for generating well-typed program constructs such as classes, methods, and fields. By contrast, PBSE is a new metadata format that uses patterns over the structure of program constructs to achieve conciseness and reusability.

Other more expressive metadata representations have been proposed in the literature. RDF [93]—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 [84]—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 [89] uses XML to represent information for managing the resources of a heterogeneous Grid, including computers, data, telecommunication, networks, and software. Orso, Harrold, and Rosenblum [102] 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. PBSE could simplify the analysis and testing of framework-based applications by capturing their architectural properties.

The complexity of enterprise metadata has motivated the creation of code generators that can automatically create metadata files based on higher level input. XDoclet, a popular open-source extensible code generator [153], is often used to automatically generate XML deployment descriptors for EJB from the source of a Java class. XDoclet works by parsing Java source files and special metadata tags (annotations inside Java comments) in the source code. Output is generated by using XDoclet template files that contain XML-style tags to access information from the source code.

The XDoclet metadata tags suffer from the same set of limitations as Java 5 annotations. Our translation tool described in Section 4.3 can be retargeted to output standalone XML deployment descriptors rather than annotated Java source files.

2.3 Reusing Additional Concerns Across Languages

Our approach to reusing additional concerns across languages is rooted in metadata translation, expressing additional concerns 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. [51] 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 [98] 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. [117] transform across different metadata formats to achieve knowledge representation compatibility in different domains by means of domain knowledge. Hernández et al. [58] translate their custom metadata specifications for database mapping and queries.

Popa et al. [108] 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 additional concern implementations declaratively.

Reusing Additional Concerns with AOP Aspect-oriented Programming [73] is the foremost programming discipline for implementing additional concerns. It has been debated which additional concerns can be treated separately [74]. However, our approach reuses only those additional concerns 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 [145] and JBoss AOP [67], can introduce metadata to programs (e.g., `declare annotation` and `annotation introduction`), thereby implementing additional concerns. 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. [95] 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 additional concerns. Instead of generating database code directly, our approach generates metadata that enables the target program to interface with platform-specific ORM systems.

DART [52] 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 [29].

Devadithya et al. [32] 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 additional concern implementations. Although our reflective capabilities are not as powerful and general, we support both Java and C++ as our source-to-source compilation platforms.

2.4 Metadata Invariants: Checking and Inferring Metadata Coding Conventions

Metadata invariants are related to validating metadata, pattern-matching, and code generation.

2.4.1 Validation for Metadata

Metadata invariants share the objective of validating the correctness of metadata with several prior efforts. Eichberg et al. [40] verify the correctness of annotation-based applications by checking the annotations' implementation restrictions and dependencies. An automated, user extensible tool reports the cases when the verified source code violates any restrictions or dependencies. Noguera et al. [99] enhance annotation declarations with meta-annotations that define various constraints to check the correctness of using annotations. The constraints, expressed as Object Constraint Language queries, must be satisfied whenever the corresponding annotations appear in the program. An automated tool validates the definitions of annotation model constraints at compile time. Cepa et al. [23] check the correctness of using custom attributes in .NET by providing meta-attributes that define dependencies between attributes. An automated tool checks these attribute dependencies declaratively expressed as a custom attribute.

Orso, Harrold, and Rosenblum [102] propose using metadata to support a wide range of software engineering tasks in the domain of distributed component-based systems. Their goal is to facilitate the process of testing and analyzing components. MIL can enable a new class of program analysis and testing techniques that focus on metadata.

Minamide et al. [97] validate XML metadata using a string analyzer. Their algorithm checks and validates metadata grammar. Compared to their approach, metadata invariants make it possible to verify how metadata relates to the program constructs it tags.

Although these approaches are quite powerful and can catch many inconsistencies of using metadata, metadata invariants explicitly codify implicit metadata programming conventions and rules. By automatically inferring metadata invariants and checking them on evolving software, a metadata invariants checker can easily identify metadata-related inconsistencies and bugs. Furthermore, the declarative nature of MIL should flatten the learning curve for the average programmer.

2.4.2 Pattern-Matching Techniques

In the domain of XML processing, techniques have been proposed [13, 10, 60, 61] to extract general XML programming patterns. However, they consider the patterns occurring in XML files, not the ones that codify how metadata relates to the tagged program constructs.

Pattern-based reflective declaration [42, 64, 63] 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, MIL uses patterns over the structure of program constructs to express metadata invariants.

The programming languages community has proposed extending Java to enable the programmer to express pattern-matching [94, 82, 86]. 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, MIL extends the notion of pattern-matching facilities to verify metadata correctness.

2.4.3 Code Generation

Code generators have been employed to automatically synthesize metadata from higher level input. XDoclet, an extensible code generator [153], can automatically generate XML deployment descriptors from special source code tags. It parses Java source files to extract special metadata tags. XDoclet templates guide the generation process that can reference program constructs as well. The XDoclet metadata tags constitute yet another metadata format, and as such, can be verified using our approach.

DART [52] automatically tests program by leveraging program analysis to generate test harness code, test drivers, and test input to dynamically analyze programs executing along alternative program paths. The generated test harness using path constraints to systematically explore all feasible program paths. Our approach also automatically generates metadata invariants in MIL, but instead of unit tests that verify the correctness of individual program methods, our approach verifies global metadata properties.

2.5 Declarative Enhancement for JavaScript Programs

TAE-JS are related to interfacing with foreign languages, supporting separation of concerns via metadata, validation of metadata, and program transformation for web applications.

2.5.1 Interfacing with Foreign Languages

TAE-JS enables JavaScript to use the metadata facilities of Java. Several prior approaches focused on interfacing with foreign languages. SWIG [9] automatically generates bindings between C/C++ code and scripting languages, including Tcl, Python, Perl and Guile. Using SWIG, C/C++ code can be invoked from a scripting language using annotated header files. Exu [18] provides bindings acrsoss multiple languages. In particular, Exu generates the language bindings that can interface Java and C++. With these approaches, the programmer is responsible for maintaining binding configurations. In contrast, TAE-JS focuses on interfacing with the metadata facilities of a foreign language and automatically maintains the correctness of the inter-language interfaces.

2.5.2 Supporting separation of concerns via Metadata

Aspect-oriented programming [73] is the foremost programming discipline for modularizing concerns (especially cross-cutting concerns). There is ongoing debate as to which concerns avail themselves to be treated separately [74], which determines the applicability of TAE-JS. Its declarative programming model follows the general AOP philosophy of treating cross-cutting concerns separately and modularly.

AOP tools, including AspectJ 5 [145] and JBoss AOP [67], can add metadata (e.g., `declare annotation` and `annotation introduction`), thus implementing concerns. Because JavaScript does not have built-in metadata, TAE-JS uses Java annotations entered by means of an special IDE.

Song et al.[131] reuse the concern implementations in established languages from emerging languages by translating metadata alongside that of the main source code. By contrast, TAE-JS expresses concern implementations within the same language declaratively.

Code generators can automatically synthesize metadata from higher level input. XDoclet, an extensible code generator [153], can automatically generate XML deployment descriptors from special source code tags. It parses Java source files to extract special metadata tags. XDoclet templates

guide the generation process that can reference program constructs as well. Unlike XDoclet, our approach uses Java 5 annotations to generate JavaScript aspects.

2.5.3 Validation for Metadata

Automated tools have been used to validate the correctness of metadata statically. Cepa et al. [23] check the correctness of using custom attributes in .NET by providing meta-attributes that define dependencies between attributes. An automated tool checks these attribute dependencies declaratively expressed as a custom attribute. Minamide et al. [97] validate XML metadata using a string analyzer. Their algorithm checks and validates metadata grammar. Metadata Invariants [130] validate both XML and Java 5 annotations by codifying naming and typing relationships between metadata and the main source code. By contrast, TAE-JS uses Java typechecking to ensure the syntactic correctness of the entered metadata.

2.5.4 Program Transformation for Web applications

Washizaki et al. [154] present an AOP framework for JavaScript, AOJS. AOJS expresses advice and joinpoint constructs in XML and weaves in aspects at runtime using a proxy-based method. Since the web applications using AOJS rely on external configuration XML files, evolving these applications may require keeping the XML files consistent with the main source code.

Kiciman et al. [70] instrument JavaScript by means of a runtime profiling techniques that rewrites the abstract syntax tree (AST). Their approach offers a dynamic instrumentation tool for Web application development. Although relying on a customized JavaScript parser can hinder portability, we may adopt a similar approach to be able to add concerns at runtime.

Lerner et al. [83] provide an AOP extension for JavaScript, integrated with a JIT compiler, whose aim is to support principled runtime adaptation. BrowserShield [114, 158] rewrite JavaScript to increase the level of security against vulnerabilities in the dynamically generated code. We can use their AOP extension with dynamic weaving instead of AspectScript to extend TAE-JS to support declarative enhancement of dynamically modified code.

Chapter 3

Enhancing Source-Level Programming Tools with An Awareness of Transparent Program Transformations

Managed languages, including Java and C#, reduce the complexity of software construction by providing portability, type safety, and automated memory management. Another reason for the widespread popularity of managed languages is that they feature multiple standard libraries and portable frameworks, whose use improves programmer productivity. In fact, it has been observed that the third-party libraries and frameworks of a typical commercial enterprise application commonly constitute the majority of the codebase [35].

Object-oriented frameworks have become an integral part of enterprise software development, as their reusable designs and predefined architectures streamline the software construction process. A major draw of modern enterprise frameworks is that the developer can write business logic components using a Plain Old Object Model (e.g., Plain Old Java Objects (POJOs) and Plain Old Common Language Runtime Objects (POCOs)), business-level application objects that do not implement special interfaces or call framework API methods. Among Plain Old Object Models, POJO-based frameworks have become mainstream in the enterprise computing community, as they improve separation of concerns, speed up development, and improve portability [115].

To provide services to application objects, a framework employs bytecode engineering to enhance their intermediate representation (i.e, bytecode or CLR), commonly at runtime. Figure 3.1 demonstrates the main steps of such framework-based development. First, the source code is compiled to an intermediate representation. Then an enhancer uses bytecode engineering [31] to add new functionality to the compiled bytecode as guided by the corresponding custom metadata. Each framework uses different metadata formats, commonly expressed using XML files or Java 5 annotations, to mark framework-related program constructs. Further, the enhancer can run as a separate tool or be integrated with a class loader. Finally, the enhanced bytecode, which differs in functionality from the original source code, is executed by the framework.

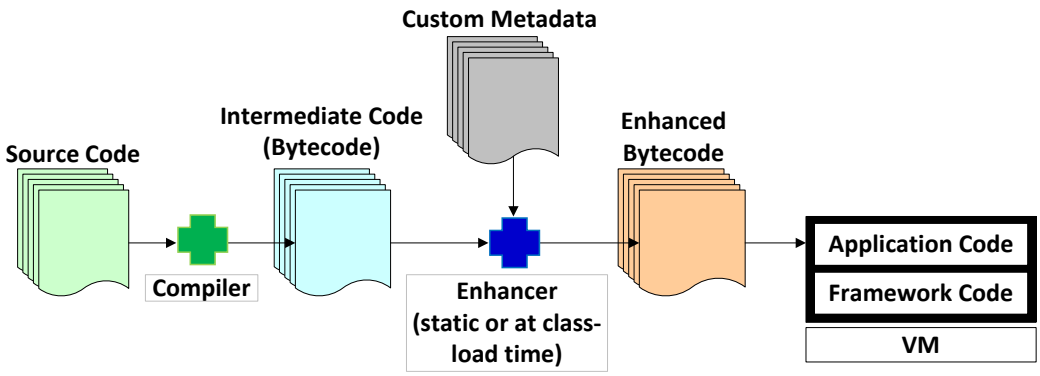

Figure 3.1: Enhancing intermediate code via bytecode engineering.

Although intermediate code enhancement has entered the mainstream of enterprise software development due to the widespread use of frameworks, no standard technique has been introduced to capture and document the enhancements. Metadata guiding the enhancement process not only is custom for each framework, but also specifies *what* program constructs are relevant for a particular framework (e.g., which fields are persistent) rather than *how* the intermediate code should be enhanced. As a result, the enhancements are, at best, documented through an informal narrative (i.e., documentation) or treated as a black box.

Lacking any formal description of bytecode enhancements, source code does not faithfully reflect the full semantics of an enhanced program. For source-level programming tools, this inability to express all the functionality of a program in source code leads to reducing the utility of those programming tools that rely on the one-to-one correspondence between the running version of a program and its source-level representation. The presence of bytecode enhancements not only hinders the ability to understand the real behavior of a program by browsing its source code, but also makes it non-trivial to map the execution of a program to its source code. The programmer may need to understand the observed behavior of a program, while relating the observed behavior of the enhanced intermediate code back to the original source and vice versa.

Although optimizing compilers have long changed intermediate representations to improve performance, the techniques developed for dealing with such optimized code (i.e., debugging optimized code) are unsuitable for dealing with enhanced bytecode. While optimizations are always semantics-preserving transformations (sometimes under certain input), enhancements change the semantics of a program in custom and difficult-to-generalize ways.

The optional Java class file's attribute `LineNumberTable` provides little value as a mechanism

for mapping enhanced bytecode to the original source code. An enhancer cannot adjust the `LineNumberTable` of an enhanced class, as the enhancements are expressed only in bytecode and have no representation in source code.

Even though Aspect Oriented Programming (AOP) [73] is often used for introducing crosscutting concerns, the AOP languages such as AspectJ [71] do not capture all the enhancements commonly introduced directly at the bytecode level, which include changing program construct names, removing program constructs, and generating new classes.

This research presents an approach that improves the utility and precision of source-level programming tools by capturing and documenting bytecode enhancements. First, we have classified commonly-used bytecode enhancements by examining the API of two widely-used bytecode engineering toolkits and by reverse-engineering the enhancements performed by two commercial enterprise frameworks. Based on our classification, we have created a declarative, domain-specific language for describing bytecode enhancements. To demonstrate the expressiveness of our language, we used it to describe the enhancements performed by several industrial and research systems that use bytecode engineering. Finally, we have used our approach to make a source code editor and a symbolic debugger enhancement-aware, thereby improving their precision and utility.

This research makes the following contributions:

- A novel approach to improving the precision and utility of source-level programming tools in the presence of intermediate code enhancements.
- The Structural Enhancement Rule (SER) language, a Domain-Specific Language (DSL) for concisely expressing structural enhancements that modern enterprise frameworks commonly apply to intermediate code.
- A debugging architecture that enables symbolic debugging of programs whose intermediate code has been transparently enhanced.

3.1 Background

In modern enterprise software development, the programmer expresses business logic components using application classes that do not inherit from special framework types or have any other func-

```
void displayMaxMortgageEligibility (Display display, double projectedIncrease) {
    double newSalary = salaryLevel.getSalary() + projectedIncrease;
    salaryLevel.setSalary(newSalary);
    double maxMortgage = calcMaxMortgage(salaryLevel,creditLevel);
    FrameField mortgageField = display.getMortgageField();
    mortgageField.setVal(maxMortgage);
    ...
}
```

Figure 3.2: Original source code.

```
void displayMaxMortgageEligibility (Display, double);
    Code:
    aload_0
    invokestatic  //jdoGetsalaryLevel;
    invokevirtual
    i2d
    dload_2
    dadd
    dstore 4
    aload_0
    invokestatic  //jdoGetsalaryLevel;
    dload  4
    invokevirtual
    aload_0
    aload_0
    invokestatic  //jdoGetsalaryLevel;
    aload_0
    invokestatic  //jdoGetcreditLevel;
     ...
```

Figure 3.3: Bytecode enhanced by the JDO enhancer.

tionality besides business logic. Then extra functionality is added to the intermediate code[1] of the application classes via bytecode enhancement to enable enterprise frameworks to provide services, thereby implementing various concerns, including persistence, transactions, and security. This development model relieves the programmer from the burden of having to implement these important concerns by hand. The programmer simply uses metadata (such as XML files or Java 5 annotations) to designate specific application objects as interacting with a framework, and all the tedious details of enabling the interaction happen entirely behind the scenes. Despite the convenience afforded by the use of such intermediate code enhancement, as we demonstrate next, its use compromises the utility and precision of source-level programming tools.

[1] In the rest of the manuscript, we use the terms *intermediate code* and *bytecode* interchangeably.

3.1.1 Transparent Persistence Frameworks

The following example comes from the domain of transparent persistence, which is used by several persistence frameworks in industrial software development, including Hibernate [8] and JDO [118]. A transparent persistence architecture combines features of both orthogonally persistent languages [6, 85, 87, 5] and data access libraries, such as Java Database Connectivity (JDBC)[142] and Open Database Connectivity (ODBC)[92].

When program data outlives the program's execution, the data is said to be *persistent*. Transparent persistence architectures provide a software framework for managing the program data marked as persistent by the programmer. The management entails synchronizing the persistent data and its stable storage representation.

A representative of a transparent persistence infrastructure for Java is Java Data Objects (JDO) [118]. JDO uses static post-compile enhancement to enable Java objects with persistence capabilities. The programmer writes Java objects to be persisted as regular Java Beans [141]. A separate JDO metafile (in XML) specifies which fields of a class should be persisted and how they map to stable storage. Finally, as specified by the metadata, the JDO enhancer adds persistence-specific methods and fields to each persistent class, enabling its instances to interact with the JDO runtime.

In particular, the enhancer changes a persistent class to implement interface `PersistenceCapable` and wraps all read and write accesses to a persistent field with special methods that interact with the JDO runtime. Thus, before the value of a persistent field is retrieved or modified, an appropriate JDO-specific action is triggered, thereby ensuring that a fresh copy of the data is retrieved from stable storage and all the changes in the application space are properly persisted.

The design of JDO satisfies the stated goal of introducing persistent capabilities transparently. JDO enables rank and file programmers to focus on *what* data is being persisted and treating *how* the data is persisted as a black box. Enterprise programmers may be aware that some bytecode enhancement is taking place, but the specific enhancements are not relevant to their primary concern—expressing the required business logic.

3.1.2 Example: Mortgage Authorization Application

Consider a mortgage authorization application used by a bank to calculate the maximum amount of mortgage eligibility, according to a set of business rules that use a customer's salary and credit

score. The application uses several transparently persistent classes, including `SalaryLevel` and `CreditLevel`, whose objects are persisted in a relational database such as MySQL or Oracle using the JDO framework.

Consider the code listing in Figure 3.2, showing a method `displayMaxMortgageEligibility`. The method displays the amount of maximum mortgage eligibility given a display object and a projected salary increase amount. As is usually the case, the method manipulates different concerns of the application: business logic and graphical user interface. Assume that the GUI part of the method contains a bug: method `getMortgageField` returns `null`, thereby causing a `NullPointerException` to be thrown in the next statement. Although the bug is in the GUI logic of the method and has nothing to do with persistence or enhancements, the JDO bytecode enhancements complicate source level debugging of the code. As an illustration, consider the enhanced bytecode of the method displayed in Figure 3.3. The programmer stepping through `displayMaxMortgageEligibility` with a standard debugger will encounter the enhancements, including various new methods, including `jdoGetSalaryLevel` and `jdoGetCreditLevel`, which can be misleading, obfuscating the location of the bug. Although tracing the enhanced bytecode with a standard debugger may accidentally lead the programmer to discover a suspect bytecode instruction, matching the instruction to the corresponding statement in the original source code may quickly turn nontrivial. Bytecode-only enhancements do not have any source code level representation.

From a different perspective, a programmer who first encounters this application with the goal of adding new features or fixing a bug would not get a realistic picture of the application's behavior by browsing the source code. In particular, the source code contains no information pertaining to the intermediate code enhancements introduced to enable transparent persistence. Thus, any change to the code could potentially lead to an unrelated change in the persistence functionality, leading to difficult-to-find errors. The metadata used to designate persistent classes only marks classes as such, but does not describe how exactly they will be enhanced.

This simple but realistic example demonstrates how transparent enhancement hinders the effectiveness of source-level programming tools. Because intermediate code enhancement has become an indispensable part of enterprise software development, new approaches are required to make source level programming tools enhancements-aware.

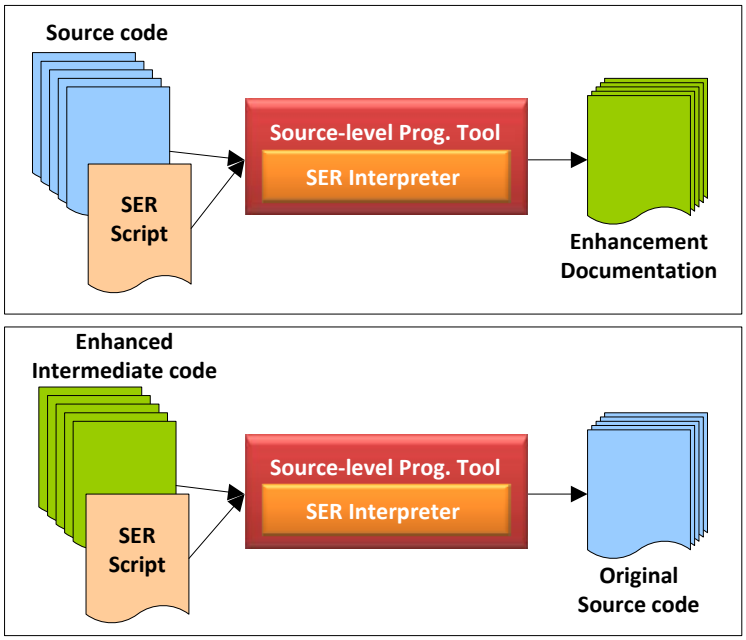

Figure 3.4: Source level programming tools with intermediate code enhancement information.

3.2 Understanding and Expressing Bytecode Enhancement

Because intermediate code is enhanced using special-purpose libraries, typically at class load time, the enhancements are poorly understood and not well-documented, if at all. The problem stems from a lack of the right expression medium for such enhancements. If intermediate code enhancements could be expressed in regular source code, there would be no need to manipulate intermediate code directly, such as with bytecode engineering. Conversely, bytecode is too low level a representation to be useful for most programming tools.

As a means of understanding and expressing intermediate code enhancements, we have introduced Structural Enhancements Rules (SER), a special purpose language for documenting bytecode enhancements. Figure 3.4 demonstrates how using SER can improve the precision and utility of source level programming tools. Specifically, the upper part of the figure shows a SER interpreter helping inform the programmer about how various program constructs will be enhanced at the bytecode level, and can be integrated with a source code editor. The lower part of the figure shows a SER interpreter being used to map the enhancements to the original source code at runtime, and can be integrated with a symbolic debugger.

Next we first explain our analysis and classification of structural enhancements. Then we describe the design and implementation of SER.

3.2.1 Structural Enhancements

Intermediate code enhancement is concerned with structural changes, which are large scale program transformations. The purpose of SER is to document structural enhancements in sufficient enough detail to improve the precision and utility of source-level programming tools that manipulate the original source code of an enhanced program. To ensure that SER provides the requisite facilities for expressing common intermediate code enhancements, we first catalog and classify common structural enhancements.

To determine what constitutes a structural enhancement, we have reverse-engineered several industrial and research systems that use bytecode enhancement.[2] In addition, we have also examined the capabilities of two major Java bytecode engineering libraries Apache BCEL[3] and Javassist [121].

Structural enhancements are the subset of general program transformations that affect the structure of an object-oriented program, including classes, methods, and fields, as well as limited changes to method bodies. Structural enhancements to method bodies are primarily confined to replacing direct field accesses with setter and getter methods as well as with other wrapper methods.

A more strict definition of structural enhancement is as follows. Modify a program, confining the set of changes to the following operations:

- Adding a new class or interface.
- Changing the type of a class or an interface (i.e., changing the parent interfaces and/or classes)
- Adding a new method or field
- Removing a method or field
- Changing the signature of a method (e.g., adding or removing parameters or changing the return type).

[2]Reverse-engineering these systems is perfectly legal, as they follow an open-source development model. Reverse-engineering enhanced bytecode turned out to be more effective than understanding the source code of the enhancers.

- Changing the type of a field
- Replacing direct field accesses with setter/getter methods

3.2.2 Semantics of Structural Enhancements

Next we provide a more formal treatment of the program transformations that constitute the structural enhancement operations commonly applied to intermediate code.

A set of original classes, $C = \{c_1, c_2, \ldots, c_n\}$
A set of added classes, $C^+ = \{c_1^+, c_2^+, \ldots, c_n^+\}$

A set of original interfaces, $I = \{i_1, i_2, \ldots, i_n\}$
A set of added interfaces, $I^+ = \{i_1^+, i_2^+, \ldots, i_n^+\}$

A set of original methods, $M = \{m_1, m_2, \ldots, m_n\}$
A set of replicated methods, $M' = \{m_1', m_2', \ldots, m_n'\}$
A set of added methods, $M^+ = \{m_1^+, m_2^+, \ldots, m_n^+\}$
A set of removed methods, $M^- = \{m_1^-, m_2^-, \ldots, m_n^-\}$

A set of original fields, $F = \{f_1, f_2, \ldots, f_n\}$
A set of added fields, $F^+ = \{f_1^+, f_2^+, \ldots, f_n^+\}$
A set of removed fields, $F^- = \{f_1^-, f_2^-, \ldots, f_n^-\}$

$\langle c \rangle p$ denotes a **class** c of program p
$\langle C \rangle p$ denotes a set of classes C of program p

$\langle i \rangle c$ denotes a **interface** i implemented by **class** c
$\langle I \rangle c$ denotes a set of interfaces I implemented by **class** c

$\langle m \rangle c$ denotes a method m of **class** c
$\langle M \rangle c$ denotes a set of methods M of **class** c

$\langle f \rangle c$ denotes a field f of **class** c
$\langle F \rangle c$ denotes a set of fields F of **class** c

ext denotes **class** inheritance relationship
$impl$ denotes **interface** inheritance relationship

Table 3.1: Syntax definitions of SER

Table 3.1 lists the symbols that we use in describing the enhancement operations, with the sets of transformed program constructs appearing first. The original program consists of a set of classes. During the enhancement, new classes can be generated and added to the program. In the original

$\texttt{AddClass}(\langle C\rangle p, \langle C^{+}\rangle p) = \cup_{i\in C}\langle i\rangle p \;\; \cup \;\; \cup_{j\in C^{+}}\langle j\rangle p$

$\texttt{AddSuperClass}(\langle C\rangle p\ ,\ \langle C^{+}\rangle p) = \cup_{i\in C}\langle i\rangle p \;\; ext \;\; \cup_{j\in C^{+}}\langle j\rangle p$
$\texttt{RemoveSuperClass}(\langle C\rangle p, \langle C^{-}\rangle p) = \cup_{i\in C}\langle i\rangle p \;\; \setminus \;\; \cup_{j\in C^{-}}\langle j\rangle p$

$\texttt{AddSuperInterface}(\langle I\rangle c\ ,\ \langle I^{+}\rangle c) = \cup_{i\in I}\langle i\rangle c \;\; impl \;\; \cup_{j\in I^{+}}\langle j\rangle c$
$\texttt{RemoveSuperInterface}(\langle R\rangle c, \langle R^{-}\rangle c) = \cup_{i\in R}\langle i\rangle c \;\; \setminus \;\; \cup_{j\in R^{-}}\langle j\rangle c$

$\texttt{AddMethod}(\langle M\rangle c, \langle M^{+}\rangle c) = \cup_{i\in M}\langle i\rangle c \;\; \cup \;\; \cup_{j\in M^{+}}\langle j\rangle c$
$\texttt{RemoveMethod}(\langle M\rangle c, \langle M^{-}\rangle c) = \cup_{i\in M}\langle i\rangle c \setminus \cup_{j\in M^{-}}\langle j\rangle c$
$\texttt{ReplicateMethod}(\langle M\rangle c) = \langle M\rangle c \mapsto \langle M'\rangle c = \cup_{i\in M}\langle i\rangle c \;\; \cup \;\; \cup_{j\in M'}\langle j\rangle c$

$\texttt{AddField}(\langle F\rangle c, \langle F^{+}\rangle c) = \cup_{i\in F}\langle i\rangle c \;\; \cup \;\; \cup_{j\in F^{+}}\langle j\rangle c$
$\texttt{RemoveField}(\langle F\rangle c, \langle F^{-}\rangle c) = \cup_{i\in F}\langle i\rangle c \setminus \cup_{j\in F^{-}}\langle j\rangle c$

$\textbf{Field}[Get \mid Set]\texttt{Replacer}(\langle F\rangle c) = \langle F\rangle c \mapsto \langle M'\rangle c = \cup_{i\in F}\langle i\rangle c \;\; \cup \;\; \cup_{j\in M}\langle j\rangle c \;\; \cup \;\; \cup_{k\in M'}\langle k\rangle c$

Table 3.2: Structural Enhancement Operations.

program, a class can implement some interfaces. An enhancer can add new interfaces to the set of implemented interfaces and can also change the super class. Methods and fields can be added to and removed from a class. Finally, existing methods can serve as templates for other methods. We refer to this operation as "replication." For example, a new wrapper method could be created based on some existing method–the new method will have the same signature, but a different name. There are no explicit constructs for changing a field or a method–this transformation can be expressed by removing the old version and subsequently adding a new one.

Table 3.2 demonstrates the semantics of structural enhancement operations using set operations. Adding new program elements to existing ones is described using $\cup$, the set union operator. Removing program elements is described using $\setminus$, the set difference operator. Replicating program elements is described using $\mapsto$ and $\cup$, which designate a new element being created based on some existing element and added to the set, but the existing element still remaining in the set.

3.2.3 SER Language Design

The Structural Enhancement Rules (SER) language is a declarative, domain-specific language that was designed to be easy to learn, use, and understand. To show how SER can serve as an effective medium for expressing intermediate code enhancements, we introduce it by example.

```
1  Program JDO Using SUPER_JDO_SER
2  Begin
3      EnhClass.Name = OrgClass.Name
4      EnhClass.AddInterface("javax.jdo.spi.PersistenceCapable")
5      Var Pattern fieldP
6          Begin
7              modifiers = "private"
8          End
9      Var Iterator fieldIter = OrgClass.Fields(fieldP)
10
11     -- add method that'll have prefix, "jdoSet" or "jdoGet".
12     EnhClass.AddMethod(FieldSetReplacer("jdoSet", fieldIter))
13     EnhClass.AddMethod(FieldGetReplacer("jdoGet", fieldIter))
14 End
```

Figure 3.5: SER Script for the JDO Enhancement.

Describing the JDO Enhancement

Figure 3.5 shows the SER script, which documents the enhancements performed by the JDO enhancer, discussed in Section 3.1. The script starts with the keyword `Program` followed by the name of the script. SER scripts can use each other by means of the keyword `Using`. The body of the script is delineated by the `Begin` and `End` keywords rather than curly braces. We have deliberately made SER look different from the C family of languages. Another distinctive feature of SER is not using semicolons to terminate program statements–each statement is expected to start from a new line. Each script makes available to the programmer two reflective objects, `OrgClass` and `EnhClass`, representing the original class and the enhanced class, respectively. This design decision is guided by the observation that bytecode enhancement either modifies an existing class or generates a brand new class, using some original class as a template. In both cases, the program in the enhanced class can be expressed as a function of the corresponding constructs in the original class. Each reflective class object has built-in attributes `Name` and `Type`, which represent their name and class type (i.e., class or interface), respectively. The statement on line 3 expresses that the enhancer will modify the original class rather than generate a brand new one.

In addition, to the attributes, the reflective class objects make available to the programmer both accessor and modifier methods. The accessor methods of SER mirror the ones in the Java Reflection API [50], with two notable exceptions. First, SER accessor methods return declarative iterators, which can only be passed as parameters to SER modifier methods as we discuss later. Second, accessor methods can accept as parameters `Pattern` objects, which describe properties of program

```
1  Program REMOTING
2  Begin
3     Module REMOTING_PROXY
4        Begin
5           Var Pattern fieldP
6           Begin
7           modifiers = "private"
8           type = REMOTING_IFACE.EnhClass.Name
9           name = "obj"
10       End
11    EnhClass.AddField(fieldP)
12    .......
13    End
14
15    Module REMOTING_IFACE
16    Begin
17    .......
```

Figure 3.6: SER script for the Remoting enhancement.

constructs. For example, the `Pattern` starting on line 5 is named `fieldP`, and it describes all the fields or methods that are `private`. The statement on line 9 uses the `fieldP` `Pattern` to obtain an `Iterator` of all the `private` fields in the original class. SER Patterns provide records describing every existing property of a program construct, including its name, type, modifiers, and signature. A `Pattern` can include any combination of records, as necessary.

The SER modifier methods can only be applied to the `EnhClass` reflective object. SER provides modifier methods for adding and deleting all the language constructs, including constructors, fields, and methods. Changing a construct can be expressed by removing it first and then adding a new construct back. The statement on line 4 documents the enhanced class modified to implement interface `PersistenceCapable`. Every modifier method can accept an iterator as a parameter, and then every element represented by the iterator will be added. Methods and fields must be explicitly added to the `EnhClass` object, even if the enhancer modifies an existing class represented by `OrgClass`. A shorthand notation `EnhClass = OrgClass` represents copying all the language constructs of the original class to the enhanced class. This shorthand is useful when the enhanced class differs from the original class only by a small margin.

SER also provides special constructs for replacing direct field accesses with setter and getter methods. To that end, SER provides methods `FieldSetReplacer` and `FieldGetReplacer`, respectively, which can be taken as parameters to the modifier method `AddMethod`. These methods take a prefix for the getter or setter method names and an iterator representing the fields, accesses to

```
Program Hibernate Using SuperHibernateSer
Begin
   -- class name that'll have new class name containing
   -- "EnhancerByCGLIB".
   EnhClass.Name = OrgClass.Name + "*EnhancerByCGLIB*"
   EnhClass.AddInterface("org.hibernate.proxy.HibernateProxy")
   EnhClass.AddInterface("net.sf.cglib.proxy.Factory")
   EnhClass.AddSuperclass(OrgClass.Name)

   Var Pattern publicP
      Begin
         modifiers = "public"
      End
   Var Itererator methodIter = OrgClass.Methods(publicP)
   Var Pattern nameP
      Begin
         name = "CGLIB*" + name
      End
   -- add method that'll have new method name
   -- containing "CGLIB"
   EnhClass.AddMethod(methodIter.CreateNewIterator(nameP))
End
```

Figure 3.7: SER Script for the Hibernate Enhancement.

which are replaced. The statements on lines 12 and 13 express the rule that JDO replaces direct field accesses with setter and getter methods, whose names start with "jdoSet" and "jdoGet", respectively.

Describing the Remoting enhancement

A SER program can be comprised of multiple modules, which can refer to each other's reflective objects. A standard SER program is likely to contain multiple modules, each representing the enhancements applied to a different class.

Figure 3.6 shows a fragment of an enhancement that transforms direct references into proxy references in order to enable their execution on a remote machine. In this enhancement, used in several research systems [104, 103, 146], for each original class, proxy, implementation, and interface classes are generated. The `Pattern` starting on line 5 refers to the type of the `EnhClass` of `REMOTING_IFACE`, which is another module in the same script. In this case, the `Name` of the `EnhClass` in the other module represents the type of the field added on line 11.

```
Program SPLIT_CLASS
Begin
   -- field to be split.
   Var Pattern _tmp
      Begin
         name = "y"
      End
   Var Iterator field2SplitIter = OrgClass.Fields(_tmp)

   Module SPLIT_MAIN_PARTITION
   Begin
      EnhClass.Name = OrgClass.Name
      -- add all fields from the original class
      Var fieldIter1 = OrgClass.Fields()
      EnhClass.AddField(fieldIter1)
      -- remove the fields that'll go to
      -- the secondary partition.
      EnhClass.RemoveField(field2SplitIter)
      ...
```

Figure 3.8: SER script for 'split a class to minimize the amount of network transfer' enhancement.

Describing the Hibernate Enhancement

As an example of a more advanced feature of SER, consider the script that appears in Figure 3.7. This description captures how Hibernate [8], another widely-used commercial persistence architecture uses bytecode enhancement. Unlike JDO, Hibernate does not modify persistent classes; instead, it creates proxy classes that extend the original persistent classes, as is expressed by the statement on line 8. Furthermore, the exact name of the created proxy classes as well as the names of their methods start with hard-coded prefixes ("EnhancerByCGLIB", "CGLIB"), appended with randomly-generated integers.

To express that the names of proxy methods are based on the names of the corresponding methods in the original persistent classes, SER introduces the ability to create a derivative iterator, given an iterator and a pattern. On line 14, an `Iterator` describing the public methods in the `OrgClass` is obtained. Then on line 15, a new `Pattern` describes adding the prefix `CGLIB*` to the name of a program construct. Finally, on line 21, the `AddMethod` method takes as its parameter the return value of method `CreateNewIterator`, which creates a new iterator by applying a pattern, `nameP`, to an iterator, `methodIter`. In effect, the one-liner on line 21 declaratively expresses that the public methods in the newly-generated `EnhClass` will have names that start with "CGLIB", followed by some random number, and ending with the corresponding method's name in the `OrgClass`.

For example, if a persistent class has a method named `foo`, the generated proxy's corresponding method could be named `CGLIB123foo`.

In essence, SER provides programming support for manipulating sets of methods, constructors, and fields. An iterator representing a set of class constructs can be obtained based on a pattern. SER has a functional feel in that SER does not allow changing iterators, but rather makes it possible to derive new iterators by applying a pattern to an existing iterator. Also, program constructs can only be added to or removed from `EnhClass`, thus simplifying the API.

Describing the Split Class enhancement

Consider the script depicted in Figure 3.8, which describes one of the enhancements required to split a class into partitions, so that only the fields used by a remote computation be transferred across the network. This enhancement entails selecting a subset of fields of the original class and placing them into a newly created class, representing the primary partition, which will be sent across the network. In SER, the selection of the required fields is accomplished by first adding to `EnhClass` all the fields contained in the original class (Line 15). And then the fields intended for the secondary partition are removed (Line 18).

SER language summary

Table 3.3 summarizes the SER programming constructs. The language follows a minimalistic design, introducing new constructs only if necessary, with the goal of making it easier to learn and understand. SER conveys the enhancements declaratively and does not have explicit conditional or looping constructs. As a result, a SER script does not contain a sufficient level of detail to be used as input for a bytecode enhancer, but it is descriptive enough to document the enhancements for source-level programming tools.

In validating the usability of SER, we have documented four enhancements used in production and research systems: JDO [118], Hibernate [8], Remoting [146], and Split Class [25, 147]. The scripts describing these enhancements in their entirety can be downloaded from the project's website [126]. In the discussion above, we have presented only fragments of these scripts to demonstrate various language features of SER. SER is an interpreted language, and its interpreter can be integrated into existing programming tools. We discuss the SER interpreter's design In Section 3.2.4, and how we used the interpreter to enhance two existing source-level programming tools in Sections 8.1.1 and 8.1.2.

Program *script-name1* [**Using** *script-name2*]
Begin
[**Module** *module-name* **Begin**...**End**]
...
End
A SER script can include other scripts and can be divided into modules.

OrgClass / **EnhClass**
Reflective class objects: original and enhanced class.

Var Pattern *pattern-name*
Begin
property="*value*"
...
End
Patterns for matching program constructs, based on their properties.

Var Iterator *iterator-name*
An iterator for a collection of program constructs.

Constructors/**Methods**/**Fields** ([*pattern-name*])
Return a collection of program constructs, [possibly matching pattern-name]

Field[**Get**/**Set**]**Replacer**(*prefix, iterator-name*)
Replace direct accesses to the fields specified by *iterator-name* with *getter*/*setter* methods starting with *prefix*, and return them as an iterator

iter-name.**CreateNewIterator**(*pattern-name*)
Create a new iterator by applying *pattern-name* to *iter-name*

[**Add**|**Remove**]**Interface**/**Superclass**/**Method**/**Field** ([*iterator-name*|*pattern-name*])
Add or remove program elements specified by *iterator-name* or *pattern-name* to or from **EnhClass**

Table 3.3: SER language constructs.

3.2.4 SER language interpretation

The purpose of documenting enhancements with a SER script is to provide a bi-directional mapping between the source code of a class and its enhanced bytecode representation. The SER interpreter can take either a Java source file or a bytecode class file as parameters, corresponding to the original or enhanced versions of a class, respectively. Another required parameter is the SER script associated with the input file. Although it would be possible for the interpreter to search for SER scripts based on the input files' names, in the current implementation it uses a configuration file that specifies which SER files work with which Java or class files. If the SER file cannot be located, the interpreter signals an error.

When processing a SER script, the interpreter parses the main script and all the included scripts specified with the `Using` keyword. Figure 3.9 demonstrates the interpreter's process flow, which differs depending on the type of the input file used to parameterize the interpreter.

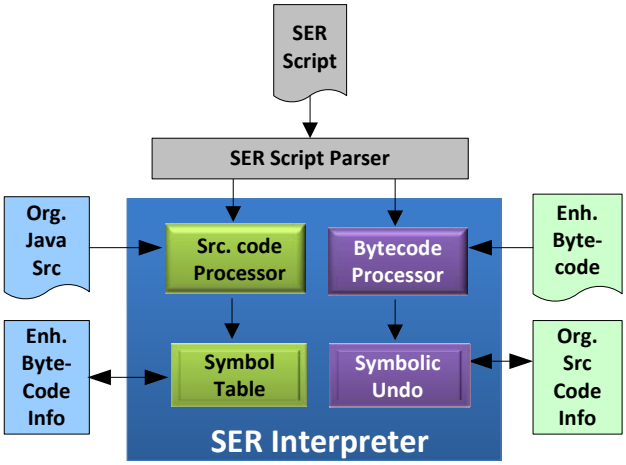

Figure 3.9: SER Interpreter.

If a Java source file, containing the original source code, is passed to the interpreter, the file is processed using the Eclipse JDT API [39]. Then the enhancement information processed by the SER parser is combined with the one of the original Java source. The result is stored into a symbol table module that implements a quick lookup mechanism capable of retrieving, in constant time, all the enhancements applied to a given program construct. Finally, an external API to the symbol table makes it possible to retrieve the enhancement information. In summary, the API uses JDT AST constructs as lookup keys to retrieve the enhancements associated with them. For example, sending a class object as a parameter will return a list of lists, containing the methods, constructors, and fields that the bytecode enhancer adds to this class. Of course, some of these lists could be empty.

If a binary class file, containing the enhanced bytecode, is passed to the interpreter, the file is processed using the Javassist library [121]. Then the enhancement information processed by the SER parser is combined with the one of the enhanced bytecode. The result is stored in a format that we call *Symbolic Undo*, which is a collection of instructions that can be used to map every enhancement back to the original source code. We will detail the exact format of Symbolic Undo when we discuss a symbolic debugger for enhanced programs that we implemented as a case study.

3.3 Summary

This chapter has argued about the value of enhancing source-level programming tool with an awareness of transparent program transformations. To enable such awareness, we have introduced SER, a declarative language that concisely describes structural enhancements. Bytecode enhancement has already entered the mainstream of enterprise software development, and future enhancers are likely to transform programs in even more complex ways. As a result, source code alone will become even less sufficient in presenting a realistic picture about the functionality of a program. Our approach of making programming tools aware of bytecode enhancements has the potential to address this problem, and help programmers enjoy the benefits of transparent bytecode enhancement without suffering any of its disadvantages.

Chapter 4

Reusable Enterprise Metadata with Pattern-Based Structural Expressions

Automatic programming is the Holy Grail of software engineering. While generating a complete program automatically from a high-level description is still a futuristic vision, the declarative programming models of modern enterprise frameworks have taken the first steps toward this vision. A programmer expresses additional concerns, including persistence, transactions, distributions, and security using metadata, and frameworks provide the requested functionality. In a typical enterprise application, programmers implement the core functionality (i.e., business logic) by writing source code and additional concerns by declaring metadata.

Metadata identifies program constructs (e.g., classes, methods, fields, etc.) as interacting with framework services, which add the requested functionality by transparently injecting it into the identified program constructs. As an example, persistence frameworks automatically persist the fields of a class annotated with the `@Column` JPA [138] annotation. Programmers only need to declare which object fields correspond to relational database columns, and persistence frameworks supply all the required functionality to render the annotated fields persistent.

Enterprise 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 J2EE [143]. More recently, metadata tags have been introduced into mainstream programming languages, including Java and C#. Java 5 annotations are a part of the source being placed near program constructs. Although in their latest releases, some enterprise frameworks have been changing their metadata format from XML to annotations, one can find examples of both metadata formats being used in modern enterprise applications.

Despite being the foundation of the declarative programming model, enterprise metadata has its share of maintenance and evolution burdens. Since XML configuration files refer to program construct names of the main source code written in Java or C#, such XML files must be kept consistent as the main program is maintained and evolved. If, for example, a name of a program

construct changes, the change must be propagated to the XML configuration file, lest the mismatch causes some framework functionality to fail.

Though annotations are part of the source code, they also hinder program maintenance. Located right next to program constructs, annotations are not affected by the changes to the constructs' names (e.g., renaming an annotated field does not affect its annotation). This tight coupling, however, is a double-edged sword—it hinders the transitioning between framework vendors that use different annotations to express equivalent functionality. For example, unit testing frameworks, such as JUnit 4 [136] and TestNG [12], use different annotations for test methods and other domain-specific information. Changing annotations scattered across an entire codebase is a tedious and error-prone undertaking, whose prohibitively high costs can cause the *Vendor Lock-in* anti-pattern [17].

Additionally, annotations cannot convey any structural information between programmer written code and framework functionality. For instance, the `@Column(name="someName")` annotation added to field "someName" only expresses that this one field and the database column to which it is persisted share the same name. Annotations cannot express that this invariant, for example, holds true for all the private fields of a class.

Finally, reuse—a major vehicle for increasing programmer productivity—is impossible with enterprise metadata, including both XML and annotations. XML descriptors are crafted individually for different classes. Annotations must be added separately not only to individual programs, but also to all the classes within the same program. Framework services are often used according to certain patterns that tend to be repetitive. For example, in many programs, all private fields may have to be made persistent. Such patterns cannot be encoded once and applied to multiple programs or even classes—each program or class must be annotated anew and its annotations maintained separately.

To address a lack of general reusability and to improve on other properties of enterprise metadata, this research presents a new metadata format—Pattern-Based Structural Expressions (PBSE). By matching the structure of a program with pattern-based declarations, PBSE captures the relationship between metadata and the program's source code, making the metadata easier to author, understand, reuse, and maintain. PBSE is not only more concise than either XML or annotations, but PBSE specifications can be easily reused across different applications.

By addressing limitations of enterprise metadata, this research presents the following novel contributions:

- A clear exposition of the advantages and shortcomings of mainstream enterprise metadata formats—XML and annotations.
- 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.

4.1 Motivation

Figure 4.1 demonstrates how modern enterprise frameworks use metadata, with the domain of transparent persistence as an example. On the upper part of the figure, class `ManagerBean` is persisted using EJB 2, a persistence framework that uses XML configuration files to specify how instances of Enterprise Java Beans are mapped to relational database tables. On the lower part of the figure, class `ManagerEJB`, with equivalent functionality to that of `ManagerBean`, is persisted using EJB 3, another persistence framework that uses Java 5 annotations as its metadata format. As we argue next, each of the two enterprise metadata formats has limitations that complicate the development and maintenance of enterprise applications that use frameworks.

4.1.1 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. Omitting some tags may result in 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.

```
public abstract class ManagerBean extends javax.ejb.EntityBean {
    public abstract String getOrderId ();
    public abstract String getStatus ();
    public abstract void setOrderId (String param);
    public abstract void setStatus (String param);
    ...
}
```

```
<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;
    }
    ...
}}
```

(2) Metadata annotations.

Figure 4.1: Transparent Persistence Framework Example.

Annotations are more straightforward than XML files, because annotations are part of the Java language. Nevertheless, adding annotations according to a convention set by a particular framework may quickly become challenging. Even though a code completion facility of a modern Integrated Development Environment (IDE) 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.

4.1.2 Understandability

Consider a programmer assigned to take over an existing codebase written using an enterprise framework dependent on metadata. Examining XML files by hand is quite tedious. XML is a computer format optimized for automated processing rather than 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, the invariant that all private fields of class `ManagerEJB` are persistent is not explicitly expressed. To determine this invariant, 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.

4.1.3 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` 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. The invariant stating that all fields that share names with database columns must be persisted is implicit and cannot be enforced through annotations.

Maintenance of framework-based applications is a formidable challenge that has been the target of several recent research efforts [116, 150, 106]. These efforts, however, could be simplified if more expressive metadata could explicitly encode dependencies that must be preserved during program evolution.

4.1.4 Reusability

Both XML and annotation-based metadata representations are not reusable. The persistence information must be explicitly encoded for each class. In Hibernate 2 [8], 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` does not work with any other class. As a consequence, the knowledge about persisting `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 leveraged in annotating other classes, due to annotations not being able to express any structural information. Each annotation provides information only about the annotated program construct. Annotations cannot express the relationships between the annotated elements (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` directly corresponds to the name of the field of the getter method it is annotating. Because this invariant is only implicit, it cannot be reused in other contexts, such as different classes or another application.

Without sophisticated automated programming tools that utilize machine learning to extract such

```
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 }}
```

(1) Java code for PBSE metadata 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 = (m.name =~ s/^get//^[A−Z]/[a−z]/)
12             Where (public * get*Id ())
13             @Column.primaryKey = true
14             m += @Id
15 MyJPA <"package1">
```

(2) PBSE metadata.

Figure 4.2: Transparent Persistence Framework PBSE Example.

implicit invariants from metadata, enterprise metadata cannot be reused systematically. Since the precision of such programming tools tends to differ widely, existing enterprise metadata remains not reusable. A new metadata format designed with reusability in mind could improve programmer productivity.

4.2 Pattern-Based Structural Expressions

By examining metadata specifications of enterprise frameworks from different domains across multiple applications, we have observed that, in most cases, metadata is not added to a program randomly, but tends to follow 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.

Consider the original motivating example from the transparent persistence domain. An equivalent PBSE specification in the lower part of Figure 4.2 is applied to the Java code in the upper part. 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 4.2 (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[1] 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 9). The **`Where`** clause on line 10 matches the getter methods following the JavaBean naming convention.[2] Line 11 attaches `@Column` metadata to the matched methods, and line 12 sets the `name` property of

[1] We use the Perl language regular expression style due to its wide adoption.

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

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 17 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 another line of code (e.g., `MyJPA<"package2">`).

4.2.1 Examples from Different Domains

The applicability of PBSE is not confined to persistence frameworks. We have discovered that enterprise 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.

JUnit

Unit testing frameworks exercise test methods designated as such using metadata. A well-known and widely-used unit testing framework is JUnit [88], 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 4.3 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.

```
Metadata MyJUnitSuite <Package p>
    Class c in p
        Where (public class Test*)
            MyJUnitTest <c>

Metadata MyJUnitTest <Class c>
    Method m in c
        Where (public void before* ())
            m += @Before
        Where (public void after* ())
            m += @After
        Where (public void test* ())
            m += @Test

MyJUnitSuite <"package1">
```

Figure 4.3: PBSE metadata for JUnit framework.

```
Metadata MyTestNGSuite <Package p>
    Class c in p
        Where (public class Test*)
            MyTestNG <c>

Metadata MyTestNG <Class c>
    Method m in c
        Where (public void before* ())
            m += @BeforeMethod
        Where (public void after* ())
            m += @AfterMethod
        Where (public void test* ())
            m += @Test
        Where (public void set* ( * ))
            m += @Parameter
            @Parameter.value = (m.name=~s/^set//)

MyTestingNG <"package1">
```

Figure 4.4: PBSE metadata for TestNG framework .

TestNG

TestNG [12] 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 technical reasons. Not changing vendors solely due to a prohibitive upgrade challenge is described by the Vendor Lock-in anti-pattern [17].

```
Metadata MySecurity <Package p>
   Class c in p
      Where (public class *)
         c += @SecureObject
         MySecureObject <c>

Metadata MySecureObject <Class c>
   Method m in c
      Where (public * get* ())
         m += @Secure
         @Secure.SecureAction = READ
      Where (public void [set|add|remove]* ())
         m += @Secure
         @Secure.SecureAction = UPDATE
      Where (static public Object+ create* ()
         m += @Secure
         @Secure.SecureAction = CREATE
      Where (public * delete* (*)
         Parameter p in m
         p += @Secure
         Where ((Object+ *))
            p += (@Secure.SecureAction = DELETE)

MySecurity <"package1">
```

Figure 4.5: PBSE metadata for security framework .

PBSE, being external to the main source code, removes this anti-pattern. Figure 4.4 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 setter 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.

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 enterprise application is divided into access control and encryption. An example of a security framework for enterprise applications is the Security Annotation Framework (SAF)[90]. SAF provides access control and encryption function-

ality, 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 4.5 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 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.

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 [132] 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 4.6 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

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

Figure 4.6: PBSE metadata for Spring Web Service.

methods is expressed in only 12 lines of PBSE. The patterns 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 5). 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.

4.3 Design and Implementation

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 of enterprise computing to come to a consensus. Thus, to make PBSE specifications immediately available to the enterprise programmer, we have implemented an automated translation tool that annotates Java source code based on its PBSE specification.

4.3.1 Language Summary

Table 4.1 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 enterprise framework.

Metadata *module_name* <[**Package**|**Class**|**Method**|**Field**|**Parameter**] *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*

Table 4.1: PBSE constructs.

4.3.2 Translator Implementation

Our translator takes as input a PBSE specification and a collection of Java classes, and annotates the classes as guided by the specification. Since PBSE borrows its metadata definitions from Java 5 annotation interface declarations, the two formats correspond to each other closely. The translation process adds annotations by matching PBSE regular expression patterns against program constructs.

Figure 4.7 demonstrates the translator's process flow, in which input files—a set of Plain Old Java Objects (POJOs) [115] and PBSE metadata—parameterize the translator, and annotated POJOs serve as an intermediate build step. Once the automatically annotated files are compiled, they can be deployed to be executed by their respective frameworks—most modern frameworks do use annotations as their metadata format.

The PBSE translator processes a Java source file by using the Eclipse JDT API [39] to traverse the

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

Table 4.2: Syntax definitions of PBSE.

abstract syntax tree of a Java class, matching PBSE regular expressions against each encountered program construct's name including that of classes, methods, method parameters, and fields. If a construct's name is matched, the construct is annotated with the annotations corresponding to the metadata guided by the **`Where`** clause of the regular expression.

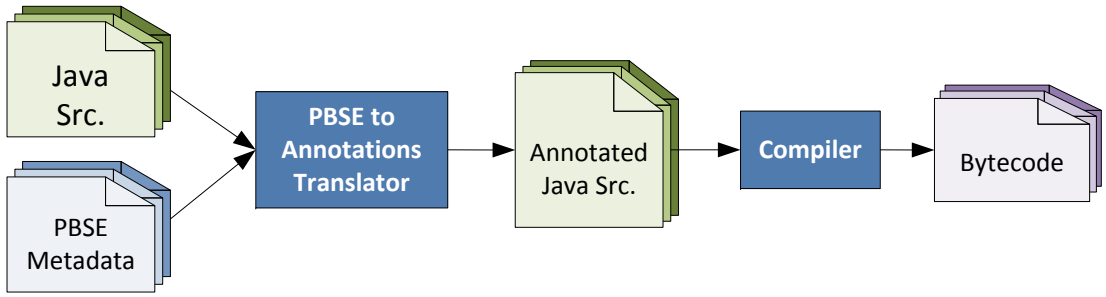

Figure 4.7: Translating PBSE to annotations as an intermediate build step.

4.3.3 Translation Semantics

Next we treat the translation process from PBSE to annotations more formally.

Figure 4.2 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

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

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

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

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

Table 4.3: Translation rules.

parameters, and fields, all of which are finite sets. Each of these structural 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 [72]. Figure 4.3 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 simplify the presentation.

4.3.4 Integration with Eclipse IDE

We have also prototyped the integration of the PBSE translator with the Eclipse Java code editor. Since PBSE metadata is entirely external to the main codebase, the programmer examining the

source code may want to be informed about how the examined program constructs are related to metadata. Following the approach initially popularized by AspectJ, whose Eclipse plug in displays visual cues about AspectJ pointcuts affecting Java program constructs, our editor shows what we call *metadata tips* that inform the programmer about the metadata attached to the examined program constructs.

4.4 Summary

This chapter have described Pattern-Based Structural Expressions, a new metadata format that offers several software engineering advantages compared to the mainstream metadata formats such as XML and annotations. PBSE leverages the common patterns between the source code and its metadata exhibited by modern enterprise framework applications. By explicitly capturing and expressing these patterns, PBSE specifications convey metadata information concisely and can be reused not only within the same application, but also across other applications that use the same enterprise frameworks. To increase the applicability of PBSE, we have implemented an automatic translation tool that annotates Java source code based on its PBSE specification. Our translation tool can also be useful for adding initial annotations to classes, even if it is not a developer's intention to use it continually. The ability to automatically add annotations to large codebases can reduce programmer burden. As enterprise software development has become heavily dependent on frameworks for implementing most of the additional concerns, the role of metadata has gained prominence. Enterprise programmers spend a substantial amount of their time and efforts implementing and maintaining metadata. It is, therefore, important to take a close look at how metadata is expressed and whether improvement is possible. In that light, this work explores these difficult questions and proposes a new metadata format that improves on the existing state of the art.

Chapter 5

Reusing Additional Concerns Across Languages

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 additional concerns, 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 [101], JRuby [24], Jython [68], and X10 [119].

Because a source-to-source compiler can only directly translate a program from the source language to the target language, the additional concern 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 additional concerns from scratch.

In this research, we present a novel approach to reusing additional concern implementations across languages. Rather than reimplement an additional concern in an emerging language, the programmer can reuse the existing target language implementations. The approach enables the programmer to specify the needed additional concern in a source language program by declaring metadata. The declared metadata is then automatically translated, so that the needed additional concern implementation in the target language can be reused. If the source language compiles to multiple target

languages, the additional concern 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 [148] 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 additional concerns in third-party X10 programs. X10 programmers express an additional concern in PBSE, which is automatically translated to the metadata required for the additional concern implementations in Java and C++.

Based on our results, this research contributes:

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

5.1 Problem Definition and Solution Overview

The programming model for implementing additional concerns is becoming increasingly declarative. To add persistence, security, or testing to an application, programmers rarely write code

```
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 5.1: 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 5.2: Metadata to persist the X10 class in Figure 5.1.

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 additional concern implementation, provided as a library or a framework. Because additional concerns are expressed declaratively through metadata, a source-to-source compiler cannot emit code for their standardized implementations. Thus, to reuse additional concern 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 additional concern 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 5.1.

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.

To that end, the programmer writes a metadata specification listed in Figure 5.2. We use Pattern-Based Structural Expressions (PBSE), a new metadata format we introduced recently [148] 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 5.2, 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 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. The

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

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++ [27]. 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 additional concerns 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 additional concerns 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 additional concerns 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 additional concern. 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.

5.2 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 additional concern API generation.

[2]Surprisingly, ODB is not an acronym.

5.2.1 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 additional concerns that expose a high level, declarative programming interface. Expressing major additional concerns—including persistence, transactions, security, and testing—through metadata has become an industry practice. 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 additional concerns 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 additional concern 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 [100]. 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 [148]. 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. Additional concern 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 5.2.3). Finally, the emerging language programmers only need to declare PBSE to add any additional concern implementation to their programs.

To increase flexibility without jeopardizing performance, we also automatically generate a special target language API for each supported additional concern implementation rather than provide a pre-defined library. Automatically generating the API also makes it possible to introduce workarounds whenever an additional concern 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.

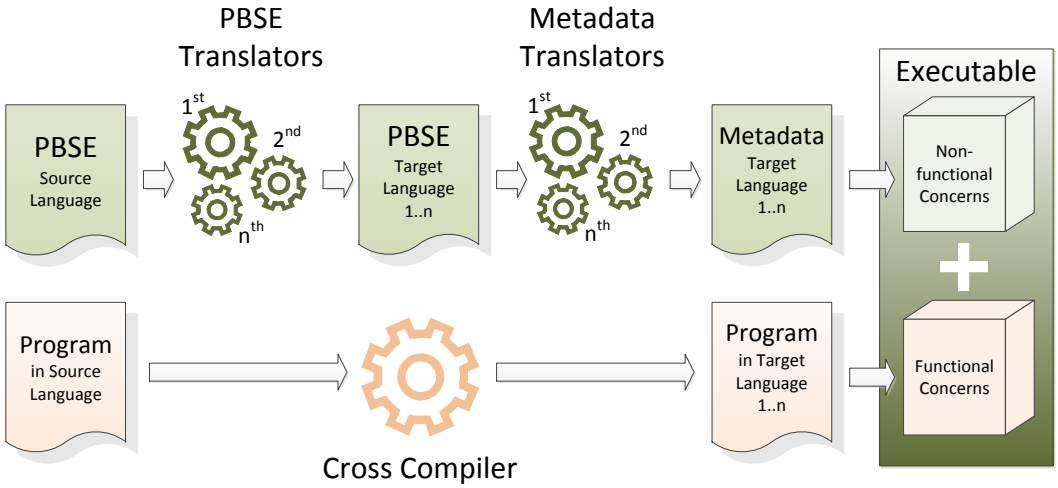

Figure 5.3: Generating Target Sources and Metadata formats.

5.2.2 Implementation Details

Figure 5.3 gives an overview of our approach. To add an implementation of an additional concern 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 additional concerns. 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 additional concern'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)[90] 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 core and additional concerns. The core concerns are implemented by translating the source program to the target one, while the additional concerns 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 additional concern is needed in both backends, the appropriate metadata has to be generated for each of them. Because language ecosystems tend to implement the same additional concern distinctly, the additional concern implementations in each compiled language may require different metadata formats and content. For example, for the persistence additional concern, a Java ORM may require XML configuration files, while a C++ ORM may require C/C++ pragmas.

5.2.3 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 5.4 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.

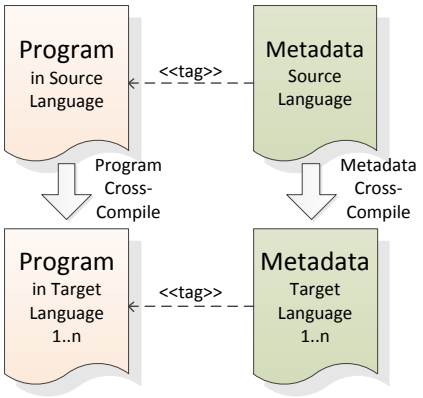

Figure 5.4: Translating Metadata formats.

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 additional concern tag the same program constructs. Across languages, the tagged source language constructs map to their source-to-source compilation targets. Because additional concern 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.

```
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 5.5: 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 + "/>"
   Where (Field f in pbse)
      @Field -> "<field/>"
      @Field.name -> "<field name=" + f.name + "/>"
      @Column -> "<column/>"
      @Column.name -> "<column name==" + f.column + "/>"
   ...
```

Figure 5.6: Meta-metadata for translating PBSE for Java to XML used by the JDO system.

To that end, our approach extends PBSE with *meta-metadata*—meta constructs that codify differences between metadata formats. In Figure 5.5, we show meta-metadata for translating between PBSE for X10 (Figure 5.2) and Java (Figure 8.19). 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.[3] The meta-metadata in Figure 5.6 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.

[3]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.

```
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 5.7: A generated visitor.

Generative visitors Based on the meta-metadata specification in Figure 5.6, our code generator synthesizes a visitor class in shown Figure 5.7. 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 5.8 was produced by one of the generated visitors.

Figure 5.9 presents a UML diagram of the visitors used in the examples discussed throughout the research. 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 5.8: Translated XML metadata for the JDO system.

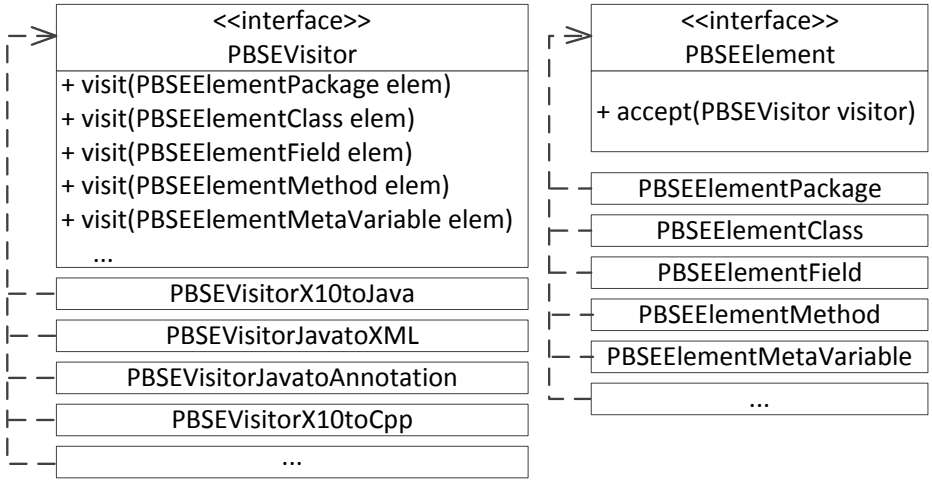

Figure 5.9: PBSE visitors translating metadata format.

Generating client APIs Since not all additional concern functionality can be expressed via metadata, some client API must supplement the automatically translated metadata process described above. To that end, our infrastructure features additional concern-specific client APIs. These APIs are invoked to access certain additional concern 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 5.10 shows

```
$[Class.name] getPersistentObj ($[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 5.10: The code template for generating database transaction API for the Java (top) and C++ (bottom) backend.

our code templates that can be used to add transactional support to persisting objects in Java and C++, shown in the upper and lower 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.

5.3 Summary

This chapter discussed a novel approach to reusing additional concern implementations across languages, this research enables emerging language programmers to take advantage of such implementations in the target languages of a source-to-source compilation process. This research

contributes an approach to reusing additional concerns 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 additional concern implementations of the mainstream language remain inaccessible to the emerging language programmers. The presented novel approach reuses additional concerns in mainstream languages by automatically translating metadata alongside compiling the source language. By eliminating the need to reimplement additional concerns in emerging languages, our approach saves development effort.

Chapter 6

Metadata Invariants: Checking and Inferring Metadata Coding Conventions

A software application comprises core and additional concerns. In a modern enterprise application, the programmer implements the core concerns (i.e., business logic) procedurally by writing source code, and the additional concerns (e.g., persistence, security, testing, etc.) declaratively by writing metadata (e.g., XML configuration files or annotations). The declared metadata configures various frameworks (e.g., ORMs, encryption and authentication controllers, etc.) that implement the required additional concerns. Although the declarative programming model cleanly separates core and additional concerns, source code and metadata interconnect so tightly that neither one can be safely evolved independently. For example, an XML tag can refer to a specific class field by name, and if the field's name changes, the XML tag must be updated accordingly. The programmer renaming a field may not even be aware that some XML configuration file references that field. Being part of the source code, annotations also may not make it clear how they should be modified when their program construct's name or type changes. For example, the name of some annotation attribute may form a naming relationship with the name of the tagged programming construct. As a result, when source code or metadata evolves, bugs can be introduced. Furthermore, these bugs would manifest themselves only at runtime.

In this research, we introduce *metadata invariants*, a new abstraction that codifies the relationships between metadata and the main source code. When metadata or the source code evolves, the metadata invariants should not be violated. A metadata invariant violation signals a potential bug that the programmer can examine further. To provide maximum benefit to the programmer, our approach automatically infers metadata invariants by analyzing extensive codebases for the relationship patterns between metadata and the source code. The programmer presented with likely metadata invariants can then either confirm their authenticity or mark them as spurious.

A metadata invariant checker can then signal when a program refactoring, enhancement, or modification violates an invariant, and alert the programmer of the possibility of a bug being introduced. In fact, we have integrated a metadata checker with the Eclipse IDE refactoring browser and ed-

```java
public class ManagerModel {
  private String orderId;
  private String orderStatus;
  ...
}
```

```xml
<class name="ManagerModel" table="MANAGERMODEL">
 <field name="orderId" column="ORDERID"/>
 <field name="orderStatus" column="ORDERSTATUS"/>
 <key column="ORDERID"/>
</class>
```

(1) XML deployment descriptor

```java
@Entity
@Table(name="MANAGERMODEL")
public class ManagerModel {
    @Id
    @Column(name="ORDERID", primaryKey="true")
    private String orderId;

    @Column(name="ORDERSTATUS")
    private String orderStatus;
    ...
}
```

(2) Java 5 Annotation

Figure 6.1: Transparent Persistence Framework Example.

itor, so that every program change is followed by checking the integrity of the original metadata invariants. We show that metadata invariants can help find those bugs that cannot be detected by state-of-the-art bug finding tools such as FindBugs [62], PMD [65], and Checkstyle [19].

To effectively express metadata invariants, we introduce a domain-specific language that we call the Metadata Invariants Language (MIL), which takes advantage of the powerful program query constructs we have developed in our prior work on improving metadata reusability [148, 131].

This research makes the following novel contributions:

- *Metadata Invariants*—a new abstraction for expressing the interconnections between the metadata and the source code of a program.
- Metadata Invariants Language (MIL)—a domain-specific language (DSL) for expressing metadata invariants.
- A practical algorithm for inferring likely metadata invariants from a codebase.
- An approach to efficiently checking metadata invariants on an evolving codebase.

- An Eclipse plug-in that adds metadata invariant checking to any Java project.
- An empirical study that assesses the effectiveness of our algorithm to infer likely metadata invariants from seven third-party, real-world enterprise applications.

6.1 Problem Definition and Solution Overview

In this section, we first present several examples of how program evolution can introduce metadata-related bugs. We then show how our approach can prevent these and similar bugs from being introduced.

6.1.1 Metadata-Related Bugs

Consider a programmer enhancing a JUnit [136] test suite with another test method. The programming conventions of JUnit 4 require that test methods be annotated with `@Test`. It is likely that having dutifully implemented the new method itself, the programmer may forget to annotate it. As a result, JUnit will not invoke this test method at runtime. If the new test method is syntactically correct, the Java compiler will not raise any errors. Furthermore, the convention of annotating methods of JUnit test suite classes with `@Test` is domain-specific, and as such its violation will not be discovered by bug finding tools that analyze programs for general bug patterns.

Consider the code snippet shown in the upper-left part of Figure 6.1. A programmer applies the *Rename* refactoring to field `orderStatus`, changing its name to `customerOrderStatus`. This refactoring seems harmless, and no refactoring precondition checker would raise any issues. Nevertheless, class `ManagerModel` happens to be mapped to a relational database table by means of a transparent persistence framework. In fact, an XML configuration file shown in the upper-right part of Figure 6.1 references this field by its original name (on line 3). As a result, after the refactoring, this field will suddenly stop being persisted to stable storage and the persistence framework in place will raise an obscure runtime error. If the programmer detects this error quickly, the bug is easy to correct by modifying the XML file accordingly. However, if the configuration in place suppresses runtime errors or displays them inconspicuously, left undetected this bug is likely to seep into production.

Annotations were introduced into Java 5 to correct some of the shortcomings of XML. The lower

part of Figure 6.1 shows how the framework transparently persist the same class `ManagerModel`, but now configured via annotations. Annotations directly tag the persistent fields, so renaming fields will no longer affect their database mappings. However, since the `name` attribute of the `@Column` annotation is a string, the programmer can easily mistype the column's name (e.g., `@Column(name="order_status")`. The code will compile, but the problem will not be discovered until the application runs. Furthermore, the programmer would have to determine the problem's source by examining the runtime exceptions thrown by the persistence framework.

The aforementioned bugs are all related to the use of metadata (i.e., XML or annotations), and as such cannot be detected by the compiler. Metadata encodes domain-specific coding conventions that lie outside of the Java language syntax. The programmer is expected to learn and follow these conventions by studying framework manuals. No tools in the standard programmer's tool chain can check the code for its compliance with such conventions.

This research presents a solution that can prevent the bugs described above. This solution consists of three parts. First of all, we observe that metadata programming conventions constitute well-structured patterns. These patterns capture how metadata tags program constructs. We call these patterns *metadata invariants* and provide a domain-specific language to express them. By statically analyzing large code bases, we automatically infer likely metadata invariants and present them to the programmer who can then either confirm or discard them. The confirmed invariants are then checked every time the program evolves, including both its source code and metadata. All violated invariants are immediately reported to the programmer, who can then determine whether the violations led to an outright bug or created some naming inconsistency that compromises the integrity of the codebase. The programmer can then take corrective actions.

6.1.2 Finding Bugs by Metadata Invariants

JUnit

Here we show how our approach can help find the bugs described above. Figure 6.2 shows the metadata invariant that codifies the metadata conventions of the JUnit 4 unit testing framework. The invariant is expressed in the Metadata Invariants Language (MIL), a domain-specific language we developed for this purpose (Section 6.2 formally presents the MIL syntax and semantics.) When designing MIL, we aimed for ease-of-learning, understandability, and conciseness. Intuitively, this invariant expresses that for all classes in a given package `p`, any class whose name ends with suffix

`TestUnit` should have all its **`public`** methods annotated with `@Test`. The specific invariant statement is expressed by means of the `Assert` statement on line 6. The `Msg` statement followed after the `:` operator will format and display an error message if the invariant's assertion is violated. For example, when our metadata invariant checker applies this invariant to a program (e.g., the checker can be run every time a source file is saved), it can report a suspected bug as follows: `MyTest.java; Line 42: method testA missing @Test`. The invariants checker by default automatically prepends the source file and line number of the violating program construct.

However, this metadata invariant does not cover all the annotations used by JUnit 4. In particular, this testing framework features `@Before` and `@After` annotations to tag the methods that setup and tear down the test suite, respectively. Fortunately, MIL invariants are strightforward to refine. Figure 6.3 shows a refined assertion on line 6 that includes all the possible annotations for **`public void`** test suite methods.

MIL can also match program constructs based on their types. Figure 6.4 shows a skeletal example of a typical JUnit 4 test suite class. On line 8, the `@Parameters` annotation tags the method returning parameterized test parameters. By convention, this method must return a type that implements the `java.util.Collection` interface. Figure 6.5 shows a MIL code snippet that matches a method's return type (line 3). Then the metadata invariant for such methods is that they should be annotated with `@Parameters`, lest a runtime error occurs.

These simple metadata invariants can be run on millions of lines of JUnit 4 tests ensuring their correctness with respect to annotations. Since unit tests are an integral part of modern software development, ensuring their integrity is paramount to improving the overall program correctness. In that light, metadata invariants fill in a unique niche of the automated bug finding tools.

```
1 Invariant JUnitAnnotations<Package p>
2    Class c in p
3    Where (* class *TestUnit)
4       Method m in c
5       Where (public void *)
6          Assert(@Test m):
7             Msg("%s missing %s", m.name, @Test)
```

Figure 6.2: Expressing JUnit 4 metadata invariants in MIL.

```
Invariant JUnitAnnotations<Package p>
   Class c in p
   Where (* class *TestUnit)
      Method m in c
      Where (public void *)
         Assert(@Test|@Before|@After m) :
         Msg("%s missing %s", m.name, "@Test,@Before, or @After")
```

Figure 6.3: MIL for JUnit 4 **void** methods.

```
...
class MyTestSuiteTestUnit {
   @Before void setUpSuite() {...}
   @After void tearDownSuite() {...}
   @Test void testMethod1() {...}
   @Test void testMethod2() {...}
   ...
   @Parameters
   static java.util.Collection myData() {...}
}
```

Figure 6.4: A JUnit 4 test suite class.

```
...
   Method m in c
   Where (m.returnType is Collection)
      Assert(@Parameters m) : Msg("%s missing %s", m.name, @RunWith)
...
```

Figure 6.5: Expressing JUnit 4 return type metadata invariant in MIL.

Hibernate

Figure 6.6 shows the metadata invariant for the Hibernate [8] framework configured through annotations. This invariant codifies a common naming convention that derives the names of persistent fields from that of their database tables. While database columns are capitalized, their corresponding fields are named according to the Java naming convention. The `Assert` statement on line 6 case-insensitively compares the `name` attribute of annotation `@Column` with the persistent field's name, thus checking a common Hibernate naming convention. The message emitted when this invariant is violated will be reported as follows: `ManagerModel.java; Line 135: order_status mismatches orderStatus`.

In isolation, such a mismatch may not constitute a bug. However, deviating from the naming

```
1 Invariant HibernateAnnotations<Package p>
2    Class c in p
3    Where (@Table public class *)
4       Field f in c
5       Where (@Column private * *)
6          Assert(@Column.name eq Uc(f.name)) :
7          Msg("%s mismatches %s", @Column.name, f.name)
```

Figure 6.6: Expressing Hibernate annotations invariants in MIL.

```
1 Invariant HibernateXMLFieldColumn<Package p>
2    Class c in p
3    Where (<class>.name eq c.name)
4       Field f in c
5       Where (<class>.<field>.name eq f.name)
6          Assert(<class>.<field>.column eq Uc(f.name)) :
7          Msg("%s mismatches %s", f.name, <class>.<field>.column)
```

Figure 6.7: Expressing Hibernate XML field column invariant in MIL.

```
1 Invariant HibernateXMLFieldName<Package p>
2    Class c in p
3    Where (<class>.name eq c.name)
4       Field f in c
5          Attribute xmlF in <class>
6             AssertExists (xmlF.name eq f.name) :
7             Msg("%s will not be persisted", f.name)
```

Figure 6.8: Expressing Hibernate XML field name invariant in MIL.

convention followed in a large codebase is a likely bug, and besides undesirable in its own right. At any rate, having received this invariant violation report while attempting to refactor the code, programmers would have to decide whether violating the invariant is warranted.

Figure 6.7 shows a metadata invariant for Hibernate configured using XML. This invariant codifies the same naming convention as in Figure 6.6, but expressed by means of XML attributes (appearing within <. . > tags). The message emitted when this invariant is violated will be reported as follows: `ManagerModel.java; Line 135; Config.hbm.xml; Line 1122: orderStatus mismatches order_status`.

The metadata invariant in Figure 6.8 verifies that all the fields of persistent classes are properly bound in the XML configuration file (e.g., the upper-right part of Figure 6.1). If the XML file does not correctly reference the persistent Java field by name, the field will not be persisted. To that end,

MIL features the `AssertExists` statement. This statement can be used only with nested iterations (i.e., **`Field`** `f in c` and `Attribute xmlF in <`**`class`**`>`), and it ensures that at least one pair of iterated items satisfies the asserted condition. The message emitted when this invariant is violated will be reported as follows: `ManagerModel.java; Line 135:  orderStatus will not be persisted.`

When this invariant is violated, its report will inform the programmer that a field will not be persisted. If that field is indeed intended to stay transient, the programmer can simply ignore this report. However, this report can prevent the *Rename* refactoring from being erroneously applied to a persistent field.

These three scenarios represent how metadata-related bugs can be introduced as a result of program evolution. By definition, metadata programming is domain-specific and reflects both explicit and implicit conventions. By verifying that evolving a program does not break these conventions, an automated checker can help prevent bugs from being introduced and can also help avoid annoying naming inconsistencies that decrease the quality of the codebase. However, because of their domain-specificity metadata programming conventions cannot be verified by means of traditional bug finding tools that apply the same common set of bug patterns to any codebase.

6.2 MIL Design

In this section, we outline the design of the Metadata Invariants Language (MIL), the domain-specific language we have created to express metadata invariants.

6.2.1 Language Summary

Table 6.1 summarizes the syntax of MIL. When designing MIL, we followed a minimalistic approach, introducing new constructs only if absolutely necessary, thus lowering the learning curve for the programmer. For example, the class iterator can iterate both through classes and interfaces of a package. MIL is a strictly declarative language, and as such lacks explicit conditional and looping constructs. Nevertheless, MIL features enough constructs to express a variety of metadata invariants of a typical modern enterprise framework.

```
Invariant name <program_construct var>
   iteration
   where
      assert
Metadata ::= [@Metadata|<Metadata>]
program_construct ::= [Package | Class | Method | Field | Parameter | Attribute]
pattern ::= [class_pattern | method_pattern | field_pattern |
            parameter_pattern | attribute_pattern]
operator ::= [is|has|eq|neq|not]
iteration ::= program_construct var in collection
   Iterate program_construct collection.
where ::= Where (pattern [operator pattern])
   Pattern-based selection.
message ::= Msg(format, arguments)
   Report a violated metadata invariant.
assert ::=
   Assert(Metadata operator pattern): message
    Verify that program_construct tagged with Metadata.
   |
   AssertExists(Metadata operator pattern): message
    Verify that at least one element of the sequence meets the condition.
[Uc|Lc](string)
    Convert all characters of the given string to upper or lower case.
```

Table 6.1: MIL constructs and grammar.

6.2.2 Assertion Semantics

Next we describe how MIL expresses how metadata invariants are to be checked by evaluating various assertions. Table 6.2 lists the symbols that describe the assertions. The sets of program's structural constructs and metadata appear first. The structure of an object-oriented program is defined by its classes, methods, method parameters, and fields; the program's metadata can be embedded in source code or written as standalone files. All of these program and metadata constructs are finite sets. Each of the program's structural constructs may potentially be tagged with metadata. Each metadata attribute is specific to the type of program construct to which it can be added, including classes, methods, method parameters, and fields. The same attribute value could potentially be used at multiple levels; for example, the `@Id` or `@Column` annotation can be applied to both methods and fields in the Java Persistence API. Each structural program construct can be matched with a declaration pattern, which are provided by replacing some substring of a construct with a wildcard character (e.g., *) that can match multiple constructs. Built-in comparison and string processing operations (e.g., `eq`, `Uc`, `Lc`, etc.) help express MIL patterns and assertions. The regular expressions of MIL **`Where`** clauses work similarly to that of AspectJ pointcuts [72].

c denotes a class	m denotes a method
f denotes a field	p denotes a parameter
t denotes a metadata	a denotes an attribute

$M_C(c)$ denotes the set of metadata of class c
$M_M(m)$ denotes the set of metadata of method m
$M_P(p)$ denotes the set of metadata of parameter p
$M_F(f)$ denotes the set of metadata of field f
$M(a)$ denotes the set of attribute of metadata t

P_c denotes a class declaration pattern
P_m denotes a method declaration pattern
P_p denotes a parameter declaration pattern
P_f denotes a field declaration pattern

$Match(e, P_e)$ denotes a declaration match of a language construct e over P_e

Table 6.2: Syntax definitions of MIL.

Table 6.3 uses set operations to express how metadata invariants assert boolean conditions for various program constructs. Specifically, an attribute value of metadata is asserted over a program construct e (i.e., class c, method m, parameter p, or field f) precisely when the construct matches a given pattern, and metadata t is attached to that pattern as specified in MIL. The presented assertion semantics does not include the `AssertExists` construct, which is somewhat of syntactic sugar and can be expressed by combining regular assert statements. Next we discuss how our invariant inference algorithms can effectively extract such domain-specific conventions and express them in MIL.

6.3 Metadata Invariants Inference Algorithm

To automatically infer metadata invariants, we have created a new algorithm that leverages global static analysis of applications that use metadata.

$$\frac{Match(c,\ P_c)\ \ (P_c,\ \frac{M(a)}{t})\ \in\ MIL}{t\ \in\ M_C(c)}\ [AssertClassMetadata]$$

$$\frac{Match(m,\ P_m)\ \ (P_m,\ \frac{M(a)}{t})\ \in\ MIL}{t\ \in\ M_M(m)}\ [AssertMethodMetadata]$$

$$\frac{Match(p,\ P_p)\ \ (P_p,\ \frac{M(a)}{t})\ \in\ MIL}{t\ \in\ M_P(p)}\ [AssertParameterMetadata]$$

$$\frac{Match(f,\ P_f)\ \ (P_f,\ \frac{M(a)}{t})\ \in\ MIL}{t\ \in\ M_F(f)}\ [AssertFieldMetadata]$$

Table 6.3: Metadata invariants assertion rules.

6.3.1 Algorithm Summary

In summary, the algorithm first scans a codebase for the presence of a naming or typing relationship between a metadata element and the program construct that the metadata element tags. Each discovered relationship becomes *an invariant candidate*. Then the rest of the codebase is analyzed to determine whether the candidate is indeed an invariant. The algorithm is tunable; it takes a threshold parameter that specifies the percentage of cases a candidate must hold true to be considered an invariant. The algorithm expresses the metadata candidates and confirmed invariants in MIL.

Algorithms 1 and 2 describe the parts of the inference process that identify invariant candidates and verify them, respectively. On line 5, Algorithm 1 iterates over all program constructs (e.g., packages, classes, methods, fields, parameters, etc.) that can be tagged with metadata (e.g., annotations, XML, etc.). Each construct tagged with metadata becomes a candidate (line 7). However, our algorithm expresses the candidates in a generalized form. Specifically, the algorithm attempts to generalize the candidates by their types and names. For the types, the candidates are generalized by finding their common supertype (i.e., field type or method return type). For example, if all the tagged methods return a class that implements `java.util.Collection`, the `ReplaceCommonSupertype` will generalize the candidates as follows: **`Where`**`(m.returnType is Collection)`. In addition, our algorithm uses the wild-card character (`*`) to generalize the candidates based on their naming correspondences. For example, if all the **`private`** fields in a program tagged with the `@Ann` annotation have the names such as `someNameAbcdef`, `someNameGhij`, `someNameKlmn`, etc., the generalization (lines 9-13) will express the field names as `someName*` in the MIL specification of their metadata invari-

Algorithm 1: `FindInvariantCandidates`

Input: **Program** p
Output: **Candidate** $C : \{c_0, c_1, .., c_m\}$

```
1  ∀ t ∈ Metadata.
2  Let pc be a program construct in p.
3
4  ForEach pc
5    If (∃pc : [pc, t]) Then
6      A : {α_0, α_1, .., α_i, .., α_n}
7      Add pc To A
8      A ← ReplaceCommonSupertype(A)
9      Do
10       l ← LongestCommSub(A, C)
11       A ← C
12       C ← ReplaceWithWildCard(pc, t, l)
13     Until l = 0
14   End
15 End
16 Return C
```

ant (e.g., **Where** `(@Ann` **private** `* someName*))`[1]. Because we only use the `*` regular expression, we can generalize the strings by continuously applying the longest common subsequence algorithm to each possible subsequence of the generalized program construct names. To use more regular expression characters, one can use algorithms for inferring regular expressions from examples [44]. Nevertheless, in our experiments we have not yet found how expanding the set of regular expression characters would improve the generalization part of our algorithm. The `*` character seems to be sufficient to express in a general form all the metadata invariants we have discovered.

In Algorithm 2, each identified invariant candidate is checked against the rest of the codebase. A candidate can either be confirmed or disproven, based on whether it holds true for the percentage of cases higher than a given threshold (line 11).

Hence, the algorithm's metadata invariant identification phases has the $O(n)$ complexity, while the confirmation phase has the $O(m*n)$ complexity, where n is the number of source code lines and m is the number of identified metadata invariants. Thus, the computational cost of inferring invariants is proportional to the number of candidate invariants discovered. The quadratic complexity may be improved on by caching the metadata-related program constructs and scanning only them during the confirmation phase. In our experiments, we have found that the algorithm rarely runs for more than a couple of minutes for codebases as large as millions LOC. Since metadata invariants are

[1]The first `*` generalizes the **private** modifier common for all the tagged fields

Algorithm 2: VerifyInvariantCandidate

Input: **Candidate** c, **Program** p, **Threshold** ε
Output: **Boolean** λ

```
1  ∀ t ∈ Metadata.
2  Let pc be a program construct in p.
3  Let α and β be the #'s an invariant candidate is
4  confirmed or disproven, respectively.
5
6    ForEach pc with t
7    If c.t = t
8      If (∃pc : [c.pc, t]) Then
9        α ← α + 1
10     Else
11       β ← β + 1
12     End
13   End
14   Return λ ← β/α ≤ ε
```

Algorithm 3: CheckingMetadataInvariants

Input: **Program** p **MetadataInvariant** μ
Output: Violations $V : \{v_0, v_1, .., v_n\}$

```
1 ∀ t ∈ Metadata; ∀ a ∈ Attribute of t.
2 Let pc be a program construct in p.
3
4 ForEach pc with t
5   If (∃pc : [pc, t] | ¬μ)) Then
6     Add pc To V
7   End
8 End
9 Return V
```

not meant to be inferred interactively, we thus far have not experienced the need to optimize this algorithm.

Algorithm 3 outlines our metadata invariant checking algorithm. In essence, the algorithm scans through the entire codebase, examining each program construct tagged with metadata. The algorithm checks each such occurrence against the input metadata invariant and collects all the violations as likely metadata bugs. In our implementation, we examine only those metadata-tagged constructs that are referenced in a given metadata invariant. For example, if an invariant refers to fields, our implementation skips class, method, and package metadata.

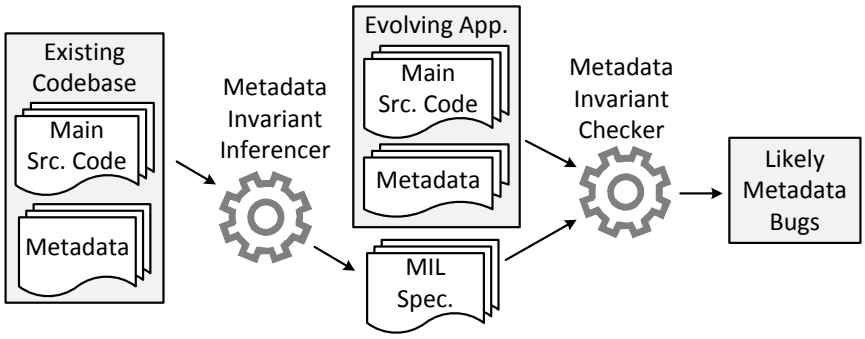

Figure 6.9: Integrating Metadata Invariants into Software Development.

6.3.2 Implementation

Figure 6.9 outlines how metadata invariants can be incorporated into a software development process. First, the metadata invariant inferencer runs on an established large code base producing invariants expressed in MIL. Then the metadata invariant checker, parameterized with the produced invariants, checks evolving applications that use the same metadata as in the established codebase. The checker reports all the invariant violations as likely metadata bugs for the programmer to examine further.

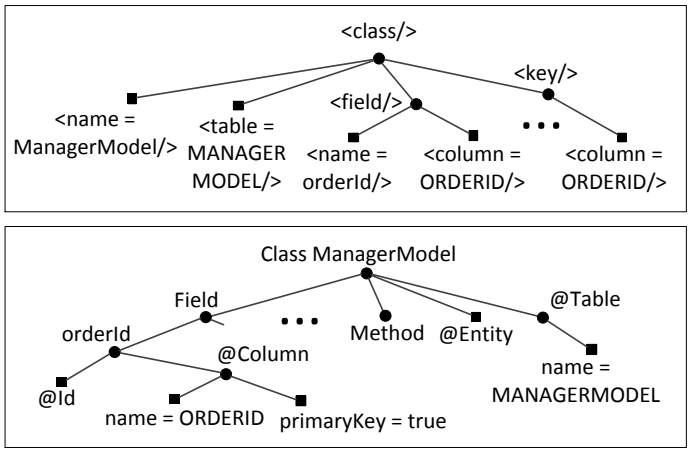

Figure 6.10: The generated tree structures for XML and Java 5 annotations (simplified version).

In our implementation, we leverage common compiler backend techniques. We walk abstract syntax trees to infer and check metadata invariants as shown in Figure 6.10. To construct such

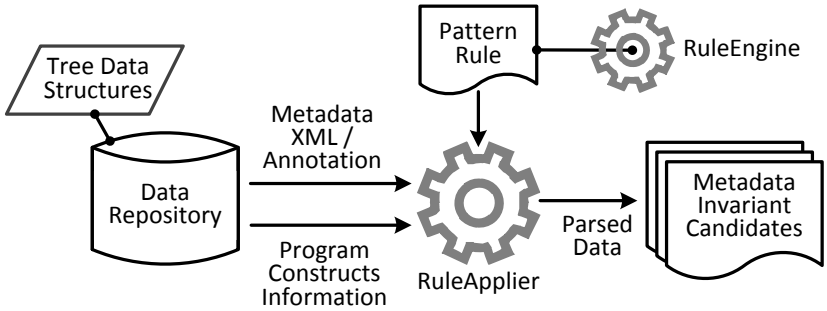

Figure 6.11: Discovering metadata invariant candidates.

abstract syntax trees, we use standard parsing infrastructures: JDT[2] for Java source files and Simple API for XML (SAX[3]) for XML. These are established technologies that significantly streamlined our implementation.

By walking the constructed abstract syntax tree, our implementation collects all the metadata related program constructs into a data repository as shown in Figure 6.11. The repository is then searched for all the correspondences between the main source code and metadata by means of a rules engine. A rules engine makes it possible to efficiently execute first-order logic rules. In particular, our implementation defines special rules to match strings based on their suffixes and prefixes exactly and case insensitively (the `RuleEngine` in the figure). The rules engine enables us to efficiently generalize the detected invariant candidates. To confirm invariant candidates, our implementation first counts the total number of invariant violations and matches, and then ensures that the violations are lower than the specified threshold.

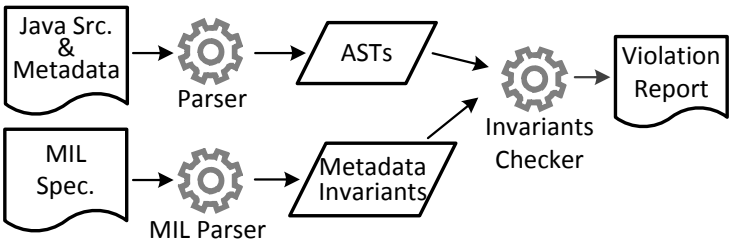

Figure 6.12: Checking metadata invariants.

Figure 6.12 outlines the backend processing required to check invariants. We use standard parsers for the main source code and metadata; we have built a custom parser for MIL. The standard parsers construct ASTs, while the MIL parser constructs AST matching patterns. Our implementation then

[2]Eclipse Java development tools – `http://www.eclipse.org/jdt`.
[3]Simple API for XML (SAX) – `http://www.saxproject.org`.

applies these patterns on the ASTs to determine where the program violates the invariants and to report the violations to the programmer.

6.4 Summary

This chapter presented metadata invariants as a new abstraction. We have demonstrated how metadata invariants can help ensure the correctness of metadata expressed in XML and Java 5 annotations. We have developed a domain-specific language for expressing metadata invariants. Our approach can infer likely metadata invariants and then check them as a program is maintained and evolved. Our inference algorithm finds the most likely metadata invariants. The inferred invariants can then be leveraged to check program correctness with respect to metadata programming in other applications that use the same metadata constructs. The applicability of our approach is not limited to Java-only, as metadata has become an integral part of modern software development. Hence, our approach can help ensure that program evolution does not introduce metadata-related bugs and inconsistencies in any application that uses metadata.

Chapter 7

Declarative Enhancement for JavaScript Programs

As Web applications now constitute an integral part of the modern computing infrastructure, the JavaScript language has become increasingly prominent. Although JavaScript was originally designed as a language for writing short, simple scripts for interactive Web pages, these days JavaScript programs keep growing in size and complexity. They often constitute multiple software modules making heavy use of standard libraries and frameworks. As JavaScript development is becoming increasing complex, following proven software engineering principles can substantially facilitate the process of engineering Web applications.

A well-known software engineering principle is separation of concerns that codifies how different facets of an application should be expressed separately to ease software comprehension and maintenance. If each concern's implementation is modularized, programmers can change concerns in isolation without perturbing the rest of the program. Recent state-of-the-art approaches add various concerns to JavaScript programs by providing libraries and frameworks that enhance JavaScript programs with persistence [22], security [133], and transactions [33].

A major drawback of these approaches is that they require modifying the original JavaScript source code by hand, creating program versions that must be maintained separately. Increasing the maintenance burden is detrimental for any software; however, it is particularly harmful for JavaScript programs, executed on a variety of client platforms in different execution environments. It is the execution environment that often determines whether a JavaScript program should be enhanced with a given concern. For example, a security enhancement may be needed in security sensitive execution environments, but would be unnecessary in other environments. Thus, there is great potential benefit in enhancing JavaScript programs transparently, on demand, without manual changes to the maintained version of the source code.

In the domain of enterprise computing, this problem has well-accepted solutions. For example, in an enterprise Java application [91], programmers can add various concerns by means of declarative annotations. A programmer can annotate some Java fields as persistent (e.g., using the JPA

[138] annotations), and a transparent persistence framework (e.g., Hibernate [8]) would render these fields persistent. Java frameworks perform all the necessary program transformations either as a static preprocessing step, at class load time, or at runtime by means of reflection. Although JavaScript has powerful reflective capabilities and the ability to change running programs through "monkey patching," declarative enhancement has not yet been explored in the JavaScript space. A key impediment is that JavaScript lacks built-in metadata that can be used to tag program constructs.

This research presents *TAE-JS* (**T**ransparent **A**utomated **E**nhancement for **J**ava**S**cript), an approach to enhancing JavaScript programs declaratively. A key novelty of TAE-JS is that it enables a host programming language to use the metadata infrastructure of another language, without modifying the host language. TAE-JS accomplishes this by means of an IDE plug-in. The multi-lingual development model of TAE-JS flexibly mixes languages during the development, but then processes them separately. As a result, the host language receives all the benefits of another language's metadata infrastructure, which includes type checking and processing APIs. The host language's syntax remains intact, so that the approach is applicable even to legacy code.

This research describes the design, implementation, and evaluation of TAE-JS and makes the following main contributions:

- An approach for using the metadata infrastructure of another language in a host language, without extending the host language's syntax.
- An approach to transparently enhancing JavaScript programs by means of generative aspects.
- Three concrete realizations of the TAE-JS approach, each featuring an annotation library and a processing plug-in, that enhance JavaScript programs with persistence, security, and transactions.

7.1 Motivating Example

Consider a code snippet in Figure 7.1 that handles user login. The entered id and password are checked at the server, with only three unsuccessful login attempts allowed. The JavaScript module containing this snippet can be used in multiple Web applications.

The number of unsuccessful attempts is stored in the variable `failedCheck`. Because this variable is

```
function checkCredentials() {
    var failedCheck = 0;
    while (failedCheck < 3) {
        enterLoginInfo();
        var id = document.getElementById('id');
        var pd = document.getElementById('password');
        if (serverCheck(id, pd))
            break;
        ++failedCheck;
}}
```

Figure 7.1: Core business logic.

```
function checkCredentials() {
    var failedCheck = 0;
    failedCheck = storage.getItem('key_failedCheck');
    while (failedCheck < 3) {
        enterLoginInfo();
        var id = document.getElementById('id');
        var pd = document.getElementById('password');
        if (! serverCheck(id, pd))
            break;
        failedCheck = storage.getItem('key_failedCheck');
        ++failedCheck;
        storage.setItem('key_failedCheck', failedCheck);
}}
```

Figure 7.2: Code (*green*) in Figure 7.1 enhanced with transparent persistence.

transient, a user having failed to enter the correct password for three times, can immediately restart the browser and continue to login. Although bypassing this constraint may be acceptable for some Web applications, one may want to prevent it from happening in high assurance domains, such as financial applications. To that end, the programmer can make the `failedCheck` variable persistent across sessions, with the persistent store being refreshed after a given timeout[1].

The functionality described above can be implemented by using a transparent persistence library. For example, Cannon and Wohlstadter [22] describe a persistence library for JavaScript programs. Figure 7.2 shows how the original code snippet can be enhanced with persistence. The variable `failedCheck` is retrieved from a storage, whose implementation is provided by a third-party library. By using the `get/setItem` library functions, the variable's state is rendered persistent.

As another enhancement, the code may need to encrypt the password before it is transferred to the

[1] If the `failedCheck` variable remains persistent forever, a user having entered an incorrect password three times will never be able to login again.

```
function checkCredentials() {
    var failedCheck = 0;
    while (failedCheck < 3) {
        enterLoginInfo();
        var id = document.getElementById('id');
        var pd = document.getElementById('password');
        if (! serverCheck(id, encrypt(pd) ))
            break;
        ++failedCheck;
}}
```

Figure 7.3: Code (*red*) in Figure 7.1 enhanced with security.

```
function checkCredentials() {
    var failedCheck = 0;
    failedCheck = storage.getItem('key_failedCheck');
    while (failedCheck < 3) {
        enterLoginInfo();
        var id = document.getElementById('id');
        var pd = document.getElementById('password');
        if (! serverCheck(id, encrypt(pd) ))
            break;
        failedCheck = storage.getItem('key_failedCheck');
        ++failedCheck;
        storage.setItem('key_failedCheck', failedCheck);
}}
```

Figure 7.4: Code (*red* and *green*) in Figure 7.1 enhanced with security and persistence.

server. This enhancement can be provided by a security library that features encryption/decryption facilities. For example, Stark et al. [133] describe an encryption library for JavaScript programs. Figure 7.3 shows how the original code snippet can be enhanced with security by encrypting the `pd` variable.

Finally, the original code snippet may need to be enhanced with both persistence and security. The resulting code appears in Figure 7.4. The code there uses both the persistence and security libraries.

The remarkable observation about this example is that applying only two concerns (persistence and security) to this code snippet created four different versions of the code that now need to be maintained separately. In fact, the number of code versions to maintain can be calculated by this formula: 2^k, where k is the total number of concerns. One can see that modifying source code by hand to enhance it with additional concerns leads to the combinatorial explosion in the number of different code versions to maintain.

7.2 Design and Implementation

In this section, we describe TAE-JS, **T**ransparent **A**utomated **E**nhancement for **J**ava**S**cript, our approach to enhancing JavaScript programs that solves the problems outlined above. First, we describe the design alternatives we have considered; then we describe our design and implementation; finally, we revisit the motivating example to show how our approach can enhance programs while cleanly separating concerns.

7.2.1 Design Considerations

The problems described in our motivating example stem from a poor separation of concerns. That is, the logic for persistence and security concerns is entangled with that of the core business functionality. This problem has been studied extensively by the aspect-oriented programming community. In the JavaScript space, an aspect extension, AspectScript [149], has been created. JavaScript programmers can write AspectScript code that would enhance the base JavaScript code with additional concerns. However, that code would have to be written and maintained by hand. As a result, aspect code would have to be manually updated whenever the original code changes (e.g., a persistent variable's name changes) or the concerns are to be added differently (e.g., another variable needs to be encrypted).

In enterprise computing, a widely used approach that effectively separates concerns is declarative programming using metadata. For example, in Java, concerns, such as persistence and security, are expressed through metadata annotations and then implemented using enterprise frameworks (e.g., Hibernate, JDO, JBoss Security, etc.). To derive the benefits of declarative programming, JavaScript needs a means to configure frameworks, and built-in metadata can significantly simplify the expression of concerns. Unfortunately, extending the JavaScript syntax with metadata would be impractical, as it would require all the JavaScript language stakeholders to come to a consensus. The practice of using external metadata, such as XML configuration files, has been repudiated in enterprise computing, with enterprise metadata nowadays expressed almost entirely by means of built-in metadata, such as Java annotations or C# attributes.

The solution presented here avails the benefits of built-in metadata to JavaScript without extending the language's syntax. In particular, TAE-JS brings the full expressiveness, type-checking, and ease-of-processing advantages of Java annotations to JavaScript programs. Furthermore, gen-

erative aspects automatically transform JavaScript code, thereby enhancing it with the specified concerns.

7.2.2 Architecture and Design

Figure 7.5 outlines the control flows of the TAE-JS approach. At the core of the approach is a specially equipped IDE. The TAE-JS IDE plug-in makes it possible to add Java annotations to JavaScript code, without modifying the latter. The JavaScript programmer selects the text of a program construct to be tagged with metadata (Figure 7.6). In response, the plug-in displays a Java annotations editor (Figure 7.7) that accepts only Java annotations, which are syntax and type checked by the Java compiler as they are being typed. The editor is configured not to save any syntactically invalid Java annotations.

The entered Java annotations are saved separately from the JavaScript code. The Java Annotation Processing Tool (APT) infrastructure is then used to processes the annotations. The APT—a part of the JDK—was created as a set of convenient APIs and a plug-in architecture that simplify the engineering of annotation processing applications in Java. TAE-JS leverages the APT to generate AspectScript code. As the final step, the AspectScript automatic preprocessor enhances the original JavaScript with the specified concerns. This automatically preprocessed code can then be included in Web applications.

The TAE-JS development model cleanly separates the roles of concern and application developers, as one can see in Figure 7.8. Concern developers create special JavaScript libraries that implement the concerns (e.g., a persistence library, a security library, etc.) Then they also create Java annotations and an APT plug-in that generates AspectScript code to add their library API calls to JavaScript programs. Next we detail each part of the TAE-JS infrastructure in turn.

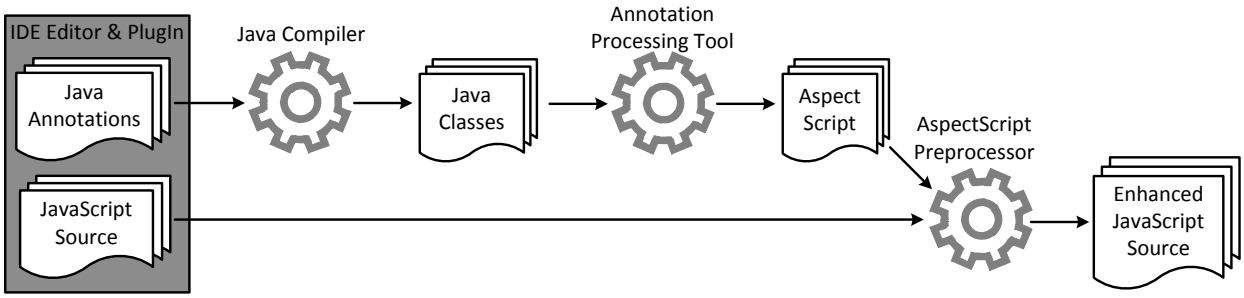

Figure 7.5: Enhancing JavaScript Programs Transparently: Control Flow Diagram.

Annotation-aware IDE plug-in

Figure 7.6 shows a screenshot of our annotation-aware IDE plug-in that makes it possible for the programmer to tag JavaScript program constructs with Java annotations. The editor creates the impression that the JavaScript syntax has been extended with Java annotations. However, a special editor is used for entering annotations. To annotate a JavaScript construct, the programmer must first select it using the mouse or the keyboard. If the selected construct can be annotated (it is a variable or a function), the IDE adds the "Annotation Editor" option to the context menu. Selecting that option invokes the annotation editor (Figure 7.7) described below. The IDE displays markers to designate every annotated construct. Hovering over a marker displays its annotation as a tooltip. In addition, the IDE provides a table view that displays all the annotations in a given JavaScript file. This view can also be used to remove annotations.

The IDE maintains the correct mapping between the tagged JavaScript constructs and their annotations in the presence of program evolution. In other words, when the annotated JavaScript program evolves, with code added, removed, or modified, the IDE keeps track of the annotated JavaScript constructs, as they move to different lines.

Finally, the IDE can, using one annotation as a sample, annotate the rest of the fields in a function. Assume, that the programmer has annotated `var failedCheck` with `@Persist (key = "key_failedCheck", variable = "failedCheck")`. Then the programmer can select a code block, containing the variables `id` and `pd`, and choose the menu option "Apply to All."

The selected fields will be automatically annotated as `@Persist (key = "key_id", variable = "id")` and `@Persist (key = "key_pd", variable = "pd")`, respectively. The IDE will automatically infer the naming correspondences between the sample variable's name and its annotation's string values (if any), generalize them, and apply the generalized naming conventions to annotate the selected variables (see Algorithm 1). Thus, our annotation-aware IDE provides all the advanced features for authoring and maintaining metadata information.

Java annotation editor

The Java annotation editor (Figure 7.7) makes Java metadata annotations available to JavaScript programmers. To allow JavaScript programmers tag their programs with Java annotations, the editor combines special UI features and automated code generation. To make the Java compiler check the syntax and type of the programmer entered annotations, the editor automatically synthesizes

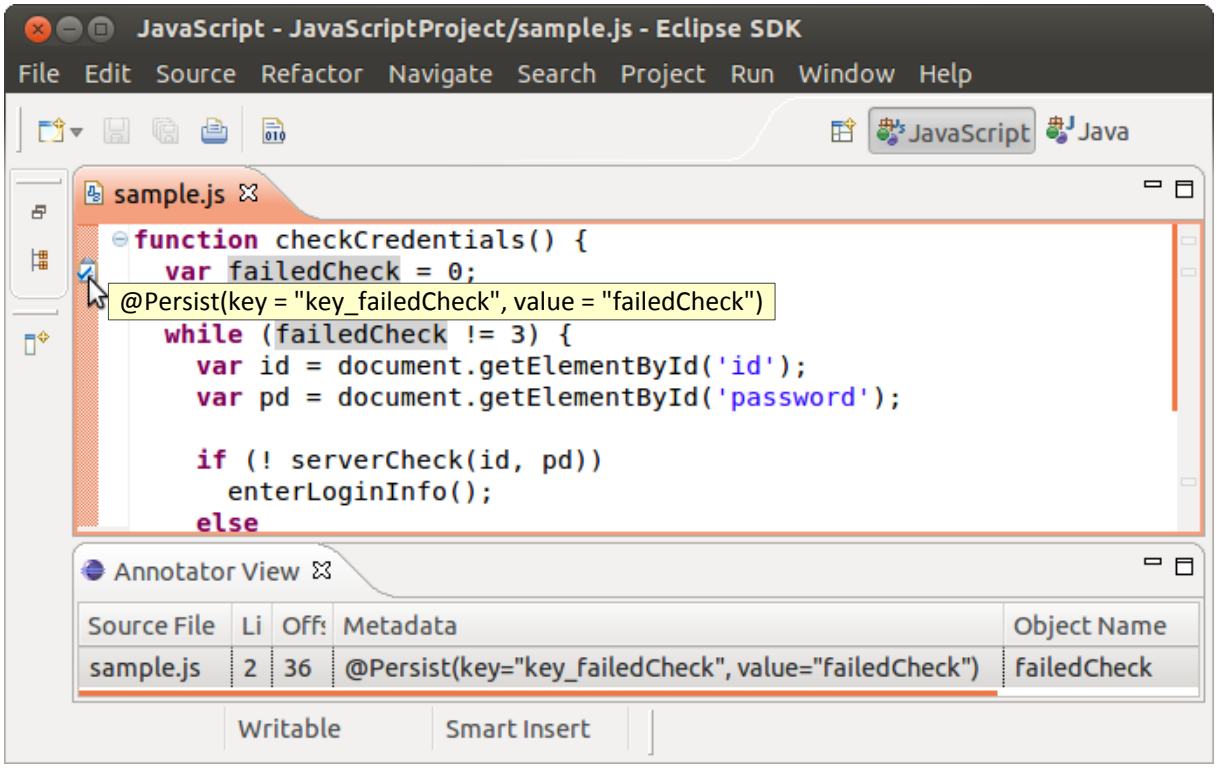

Figure 7.6: Annotation-aware IDE.

Java identifiers, which are then rendered invisible (and non-editable) to the JavaScript programmer. The synthesized Java identifiers also encode the structural information about the annotated JavaScript constructs. As a result, when processing the annotations, the APT plug-ins no longer need to refer to the original JavaScript code to generate the AspectScript aspects to transform it.

As an example, consider the code in Figure 7.9 that is handled by the Java annotation editor shown in Figure 7.7. Depending on its purpose, the code's sections can be visible or invisible as well as automatically generated or programmer entered. The code in gray is automatically generated and rendered invisible to the JavaScript programmer; this code creates a valid compilation context for the Java compiler to enable the syntax and type checking for the programmer entered annotations. To provide proper documentation, a skeletal representation of the tagged JavaScript code is shown as part of a JavaDoc comment, so an HTML document can be generated showing all the annotated JavaScript code blocks. The code in blue provides the instructions for the JavaScript programmer entering annotations. Finally, the code in red is the programmer entered annotation.

Notice that the generated Java code provides sufficient information about the annotated JavaScript

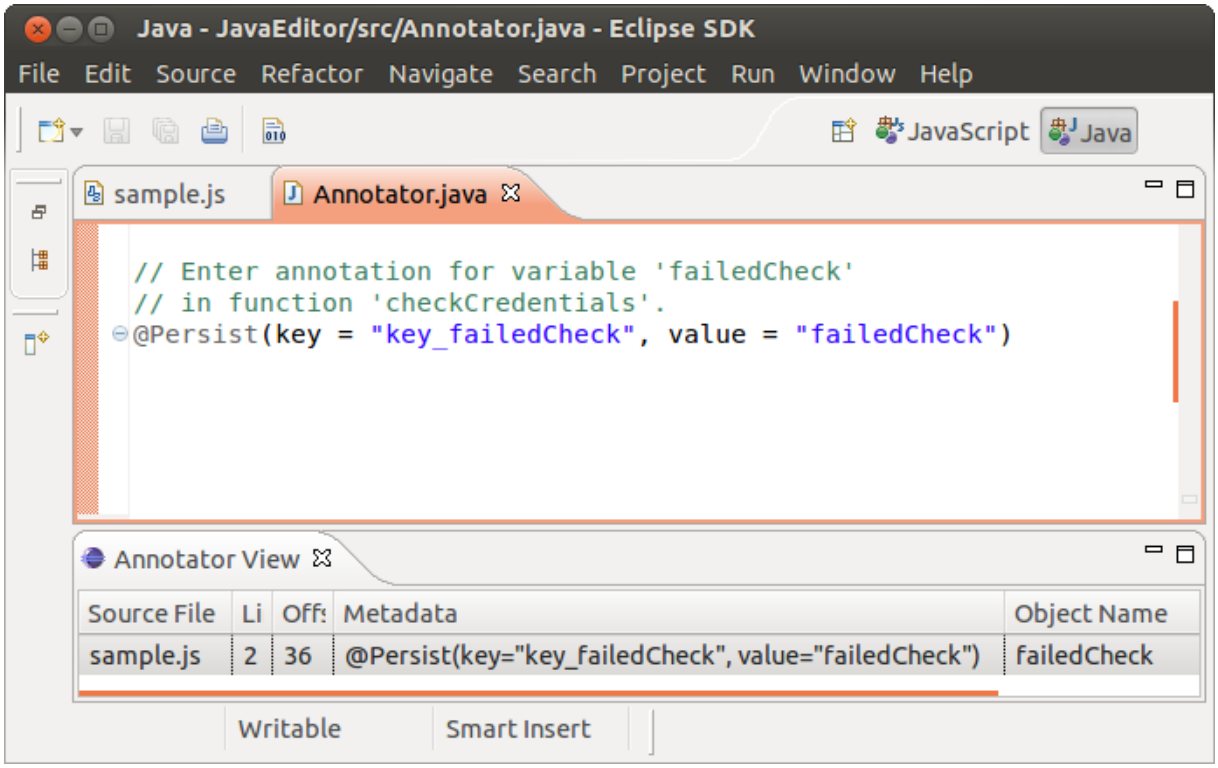

Figure 7.7: Annotation editor: annotating `failedCheck`.

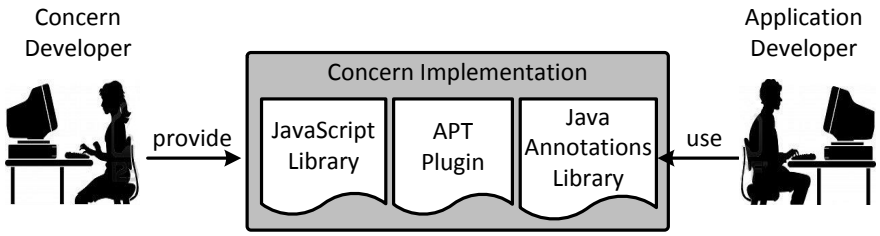

Figure 7.8: The roles played by concern and application developers in TAE-JS.

programs, so that the APT plug-ins can generate AspectScript code without having to refer back to the original JavaScript code. To that end, TAE-JS maps the JavaScript type system to the Java type system. Specifically, JavaScript functions and variables are mapped to Java classes and member fields. Functions in JavaScript and classes in Java map to each other one-to-one. The automatically synthesized Java code follows an established coding convention (i.e., function names are preceded with the `f_` prefix, and variable names with the `v_` prefix).

Algorithm 1: FindPattern

Input: Program Construct PC and Annotation N
Output: Pattern Found

```
1  Let P be a set of pattern candidates.
2  P : {p_0, p_1, .., p_i, .., p_n}
3  Attributes = getAttributes(N)
4  ForEach attr in Attributes
5    Tokens ← GetTokens(PC)
6    ForEach token in Tokens
7      P ← LongestCommSubStr(token, attr)
8    End
9  End
10 Sort(P)
11 Return p_0
```

```
1  class f_checkCredentials {}
2
3  class Annotator {
4      // Enter annotation for variable "failedCheck"
5      // in function "checkCredentials".
6      @Persist(key = "key_42", variable = "failedCheck")
7      f_checkCredentials v_failedCheck;
8      /** This Java declaration encodes JavaScript code below:
9      * <p><code><pre>
10     * function checkCredentials {
11     *       var failedCheck;
12     * }
13     * </pre></code>
14     */
15 }
```

Figure 7.9: Code in Java annotation editor; gray—invisible generated; blue—visible generated; red—programmer entered.

Generating aspects

As shown in Figure 7.8, TAE-JS requires that a concern developer provides a JavaScript library implementing the concerns, a library of Java annotations for applying the concern, and an APT plug-in to add the concern's library calls to the enhanced JavaScript code. The APT architecture provides an intuitive Java API for writing annotation processing plug-ins. A plug-in reads annotated Java classes and extracts the annotated constructs and their annotations.

Figure 7.10 shows pseudo-code for an APT plug-in for a transparent persistence library. Upon en-

```
1 class PersistAnnotationProcessor implements AnnotationProcessor {
2     ...
3     public void process() {
4         // For each declaration annotated with @Persist retrieve the annotated
5         // construct's type and name e.g., f_checkCredentials v_failedCheck;
6 
7         // Generate AspectScript code referring to function "checkCredentials"
8         // and variable "failedCheck".
9 }}
```

Figure 7.10: Pseudo-code for the APT persistence plug-in.

countering the annotation `@Persist (key = "key_failedCheck", variable = "failedCheck")` applied to the generated field named "`f_checkCredentials v_failedCheck`", the plug-in code can parse the field's name to determine that the specified JavaScript code is variable `failedCheck`, defined in function `checkCredentials`. To insert the required persistence functionality, TAE-JS generates aspects.

As our aspect language, we chose AspectScript [149], an AOP JavaScript extension that works with all the major Web browsers, including Mozilla Firefox, Safari, Chrome, and Opera. Inspired by the design AspectScheme [37], AspectScript focuses on supporting the unique features of JavaScript, including first-class functions, dynamic typing, and prototype-based programming. AspectScript features pointcut-advice mechanisms, providing all the major facilities one can expect in a modern aspect language extension.

The snippets of AspectScript code in Figure 7.11 transparently persist variable `failedCheck`. AspectScript defines pointcuts, program locations at which additional functionality (i.e., advise) should be interposed, as regular JavaScript variables. The pointcut variables `pcGetV1/SetV1` define the locations at which variable `failedCheck` is read and written, respectively. The `AROUND` and `AFTER` advice directives express that the execution of aspect code should take place in relation to the pointcuts. Anonymous functions referenced by variables `persistAdviceGetV1` and `persistAdviceSetV1` are the advice code that AspectScript interposes with the original JavaScript code.

In essence, the around advice replaces are the memory reads of variable `failedCheck` with retrieving it from persistent storage, provided by the persistence library in place. The after advice stores to persistent storage the updates to `failedCheck`. Recall that the persistent values can be set to expire after a given timeout, thereby eventually allowing users to continue trying to login.

Using the persisting of variable `failedCheck` as an example, Figure 7.12 demonstrates how program

```
var pcGetV1 = AspectScript.Pointcuts.get("failedCheck");
var pcSetV1 = AspectScript.Pointcuts.set("failedCheck");

var persistAdviceGetV1 = function(jp) {
    return storage.getItem("key_failedCheck");
};

var persistAdviceSetV1 = function(jp) {
    storage.setItem("key_failedCheck", jp.value);
};

var aspectGetV1 = AspectScript.aspect(
    AspectScript.AROUND, pcGetV1, persistAdviceGetV1);
AspectScript.deployOn(aspectGetV1, checkCredentials);

var aspectSetV1 = AspectScript.aspect(
    AspectScript.AFTER, pcSetV1, persistAdviceSetV1);
AspectScript.deployOn(aspectSetV1, checkCredentials);
```

Figure 7.11: AspectScript to add persistence.

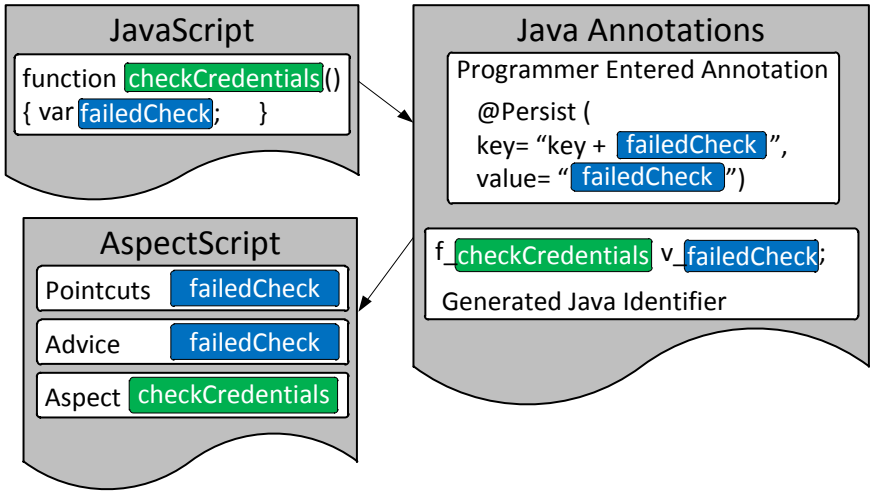

Figure 7.12: A TAE-JS data flow diagram: persisting variable `failedCheck`.

construct names flow between JavaScript, Java annotations, and AspectScript code. The function name "checkCredentials" and the variable name "failedCheck" flow from JavaScript to the generated Java code (and are also referenced in the `@Persist` annotation's attributes), and finally appear in the generated AspectScript code that transforms the original JavaScript program.

Notice, however, that this automatically generated aspect code may not be sufficient to set in place the persistence policy required for our motivating example. In particular, the programmer

```
var persistAdviceSetV1 = function(jp) {
   if (storage.getItem("key_failedCheck") < 3)
      storage.setItem("key_failedCheck", jp.value);
};
```

Figure 7.13: Hand-modified AspectScript code to add application-specific logic.

may want to invalidate the reassignment of variable `failedCheck` to 0 if the persisted value of the attempted login attempts has reached three. To that end, the programmer can easily modify the generated AspectScript code by hand as shown in Figure 7.13. Even adding this specialized functionality does not require changing the original JavaScript code by hand.

7.2.3 Template-Based Code Generation

As one can see, TAE-JS relies on generating potentially large quantities of AspectScript code. Although any code generation method can be plugged-in as part of the TAE-JS infrastructure, the reference implementation leverages template-based code generation to avoid the inconveniences of maintaining hand-crafted, ad-hoc code generators. In the context of TAE-JS, we found that a template-based approach strikes the right balance between simplicity and expressiveness. In particular, template-based code generation reduces the possibility of introducing syntax errors into the generated code, while being easy to learn, use, and maintain.

For the case studies, we used the popular StringTemplate template engine, whose design is based on the model-view-controller architecture [105]. This design separates the data used to drive code generation (i.e., the model), from the actual template (i.e., the view). The StringTemplate language includes elements of functional languages such as side effect-free expressions, independent expression evaluation, and operations on lists of objects.

Our code generation infrastructure makes several pre-defined templates per concern available to the enhancement library developer, including skeletal definitions of aspects of persistence, security, and transactions. At the API level, these entities are represented as variable, function, iterator, and AspectScript keywords. The developer can provide StringTemplate definitions for the variables, functions, aspects, pointcuts, contained in these skeletal definitions. StringTemplate also makes it straightforward to write the generated AspectScript source code to a file.

Figure 7.14 shows a fragment of the StringTemplate template that we used to implement our persistence enhancement. Our experiences with template-based code generation indicate that using

```
var pcGetV$itr$ = AspectScript.Pointcuts.get("$variable$");
var pcSetV$itr$ = AspectScript.Pointcuts.set("$variable$");

var aspectGetV$itr$ =
    AspectScript.aspect(AspectScript.AROUND,pcGetV$itr$,persistAdviceGetV$itr$);

AspectScript.deployOn(aspectGetV$itr$, $function$);

var aspectSetV$itr$ =
    AspectScript.aspect(AspectScript.AFTER,pcSetV$itr$,persistAdviceSetV$itr$);

AspectScript.deployOn(aspectSetV$itr$, $function$);

var persistAdviceGetV$itr$ = function(jp) {
    return storage.getItem("$key$_$variable$");
};

var persistAdviceSetV$itr$ = function(jp) {
    storage.setItem("$key$_$variable$", jp.value);
};
```

Figure 7.14: A portion of a StringTemplate template for generating AspectScript code to add persistence.

```
<target name="jar-persist">
    <jar destfile="persist_apt_plugin.jar" > ... </jar>
</target>
<target name="jar-security">
    <jar destfile="security_apt_plugin.jar" > ... </jar>
</target>
<target name="jar-transactions">
    <jar destfile="transactions_apt_plugin.jar" > ... </jar>
</target>
```

Figure 7.15: An Ant build script with TAE-JS rules.

this principled approach indeed reduces the possibility of introducing subtle syntax errors and accommodates the reuse of code generation functionality across different libraries.

7.2.4 Motivating Example Revisited

Recall that in our motivating example, a piece of JavaScript code (Figure 7.1) had to be enhanced with two concerns (persistence and security), potentially creating four different codebases to be maintained separately. By using TAE-JS, one can avoid branching the codebase. The JavaScript

programmer first would annotate the JavaScript codebase with the annotations to add both the persistence and security concerns. Then the concerns can be added as needed through build configuration.

In modern software development, automated tools handle the build process (e.g., make or Apache Ant[2]). These tools are configured through a script that includes the steps the tool must go through to build a software product. TAE-JS includes two additional steps that can be easily added to any major build script. The first step generates AspectScript code by running aspect generation APT plug-ins. The second step transforms the original JavaScript code to include the specified concerns by means of the AspectScript preprocessor executing the generated aspect code. The flexibility of TAE-JS lies in its ability to flexibly create the versions of JavaScript code containing the concerns required for a given deployment scenario. To that end, build managers need only to include or exclude TAE-JS APT plug-ins. As an example, Figure 7.15 shows how the TAE-JS steps can be integrated with an ANT build script. Because it is the build tool that adds the required concerns through automated program transformation, the original JavaScript codebase remains intact, thereby reducing the costs of software maintenance and evolution.

7.3 Summary

In this chapter, we presented **T**ransparent **A**utomated **E**nhancement for **J**ava**S**cript (TAE-JS), a novel approach to enhancing JavaScript programs with additional concerns. As JavaScript has become the lingua franca of Web applications, solid software engineering principles should be applied to the development and maintenance of JavaScript programs. The ability to separate concerns cleanly can particularly benefit those JavaScript codebases that may need to be reused in applications with different concern requirements. The main novelty of TAE-JS lies in embedding the metadata infrastructure of a foreign programming language in a host language, without modifying the host's language syntax. In the reference implementation of TAE-JS, we demonstrated how Java annotations can be fully utilized by JavaScript programs. Another novelty of TAE-JS is in applying generative aspects to JavaScript programs. Our results indicate that JavaScript programs can be enhanced transparently based on declarative specifications, thereby improving the overall separation of concerns.

[2]Apache Ant – `http://ant.apache.org/`

Chapter 8

Applicability and Case Studies

This chapter argues that this dissertation explores algorithms, techniques, and tools for supporting reuse and evolution in metadata-driven software development that can be a valuable to increase the programmer's productivity as well as save their efforts and time. The content of this chapter, as its title suggests, is described as followings: evaluating the applicability of the structural enhancements-awareness, validating the advantages of PBSE by comparing existing mainstream metadata formats, discussing the applicability of reusing four additional concerns across both multiple domains and languages, assessing the effectiveness of our metadata invariant inference algorithm and checking mechanism, and measuring the the programming effort incurred by the TAE-JS approach for each added concern on a Web application running in a browser.

8.1 Case Study: Enhancements-Aware Programming Tools

To evaluate the applicability of this research, we have augmented two existing source-level programming tools with an awareness of bytecode enhancements. Specifically, we have added a new browsing view to a widely-used source code editor to present the bytecode enhancements applied to the program constructs of the displayed compilation unit. We have also created a new debugging architecture that enables a symbolic debugger running an enhanced program to display the original source code, undoing the enhancements at runtime.

To check the effectiveness of the enhancements-aware tools, we have applied both of them to four different applications, each using a different bytecode enhancement scheme. The first two applications use transparent persistence architectures, albeit with drastically different enhancement strategies. While JDO modifies persistent classes, Hibernate generates proxy classes that inherit from them. Another difference between JDO and Hibernate is the time when the enhancement takes place: while JDO enhances persistent classes as a static post-compilation step, Hibernate generates proxy classes at class load time.

Two other applications come from the domain of distributed computing. The first application

performs an RMI Remoting enhancement, in which the original, local class is rewritten into a remote implementation class, and new proxy and RMI remote interface classes are generated. This rewrite is performed by several systems designed to make distributed computing in Java more intuitive, both at the source level such as JavaParty [107], and transparently at the bytecode level such as J-Orchestra [146].

Another enhancement from the distributed computing domain modifies the structure of a data class to optimize the network transfer of its instances. Class fields are divided into a set that is used by a remote server and the one that is not, and the class is rewritten into two partitions containing those sets, using the Split Class binary refactoring [147]. Finally, the partition to be transferred across the network is made to implement `Serializable` to enable the marshaling of its instances. Because the rewrite adds a new capability, it is classified as an enhancement.

We expressed each enhancement scheme in SER. For the transparent persistence architectures, we reverse-engineered the enhancements by comparing the original and enhanced versions of multiple application classes. In the distributed application cases, we simply documented in SER the transformations informally described in the respective research publications [146, 147].

8.1.1 Source Editor with Zoom-in-on-enhancements View

Intermediate code enhancement introduces new functionality behind the scenes, transparently to the programmer. Furthermore, leaving the programmer unaware of the specific changes applied directly to the bytecode has been recognized as a key benefit of the technique—it improves separation of concerns, with the programmer being responsible for business logic only. Nevertheless, as we argue next, making the bytecode enhancement information accessible to the programmer can yield software engineering benefits. For example, bytecode enhancers follow a certain convention in choosing the names of program constructs that they add. In particular, JDO starts the names of all the added setter/getter methods with prefix `jdoSet/Get`. There is nothing, however, that would prevent the programmer from writing a method starting with these prefixes, thus creating a difficult to diagnose name clash. By examining the enhancements that will be applied to a class, the programmer can quickly identify such inappropriately-named program constructs. As another example, consider the restriction of Hibernate of not being able to persist instances of `final` classes and any classes that have `final` methods. This restriction seems completely arbitrary, unless the programmer can examine how Hibernate enhances the persistent classes. This restriction becomes immediately apparent, as soon as the programmer observes that persistence-enabling proxies gen-

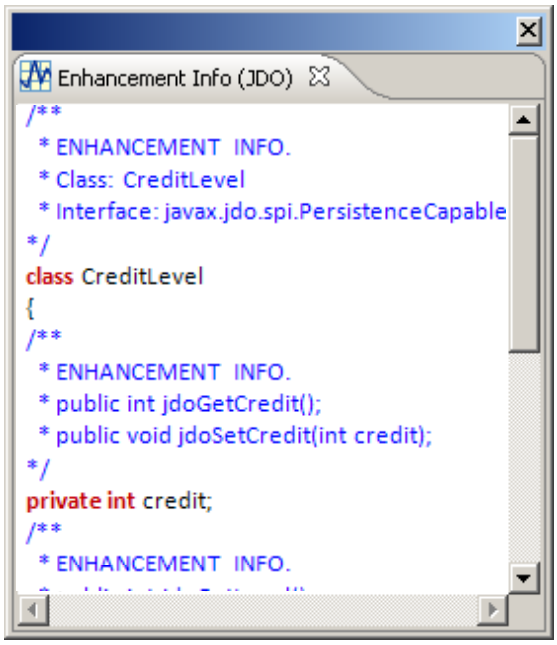

Figure 8.1: Documentation for the JDO Enhancement.

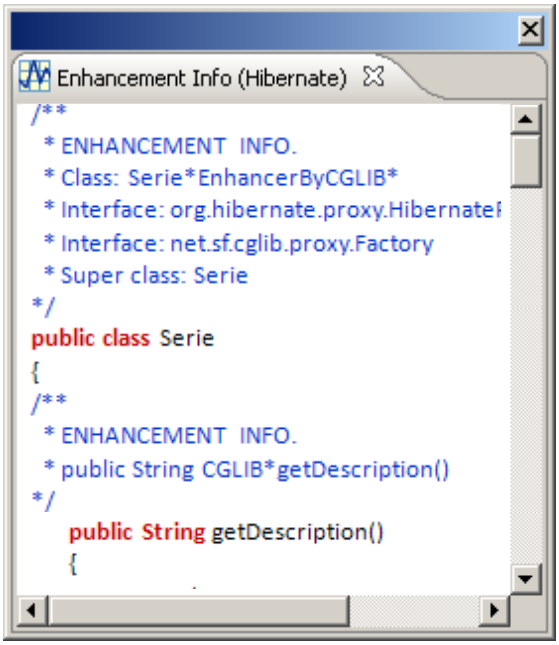

Figure 8.2: Documentation for the Hibernate Enhancement.

erated by Hibernate extend the persistent classes. As yet another example, certain performance bottlenecks in the enhanced code can be identified by examining the enhancements. For example, the use of RMI in remoting enhancements can be detrimental to performance in some networking environments. As a final example, some bugs in metadata, which describes which program constructs are to be enhanced, can be more easily identified if the enhancement code is visible.

In this case study, we have integrated the SER interpreter with an Eclipse Java code editor. Whenever the programmer selects a class whose bytecode is subject to enhancement, a new view pops up displaying an abbreviated description of the enhancements. We call this view window a *zoom-in-on-enhancements view*. Following the Eclipse tooling strategy, the view is visible only if activated, so if they so choose, the programmers are free to remain oblivious about the nature and specifics of enhancements. In the case if the original class is modified at the bytecode level, the enhancements are shown as special comments in the main editor. Since not all the information about the enhancements in known at source edit time, the enhancements cannot be expressed in source code form. If a single class is associated with several classes created during an enhancement, each of the created classes is displayed in a separate view.

Figures 8.1, 8.2, 8.3, and 8.4 show the screen-shots of the *zoom-in-on-enhancements views*, which

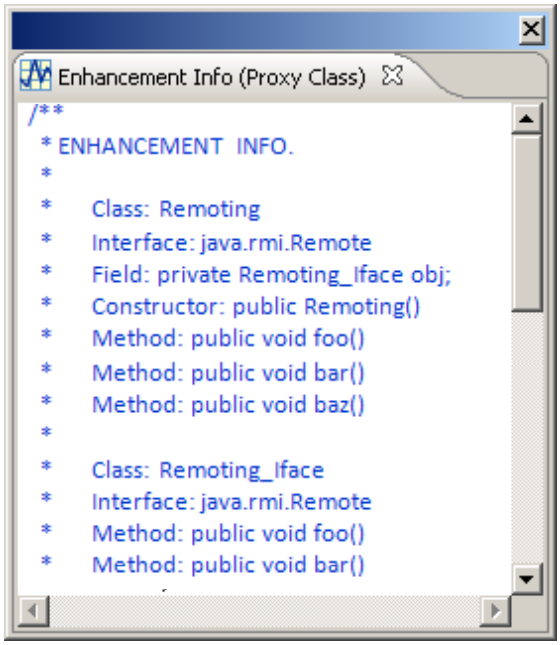

Figure 8.3: Documentation for the Proxy class Enhancement.

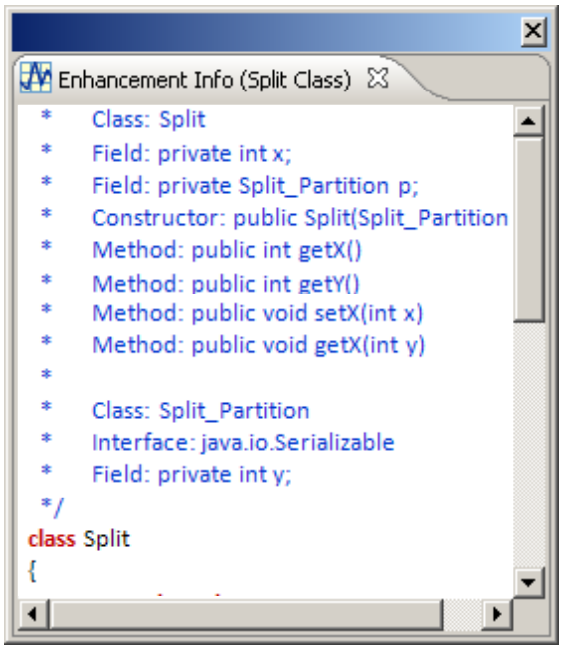

Figure 8.4: Documentation for the Split class Enhancement.

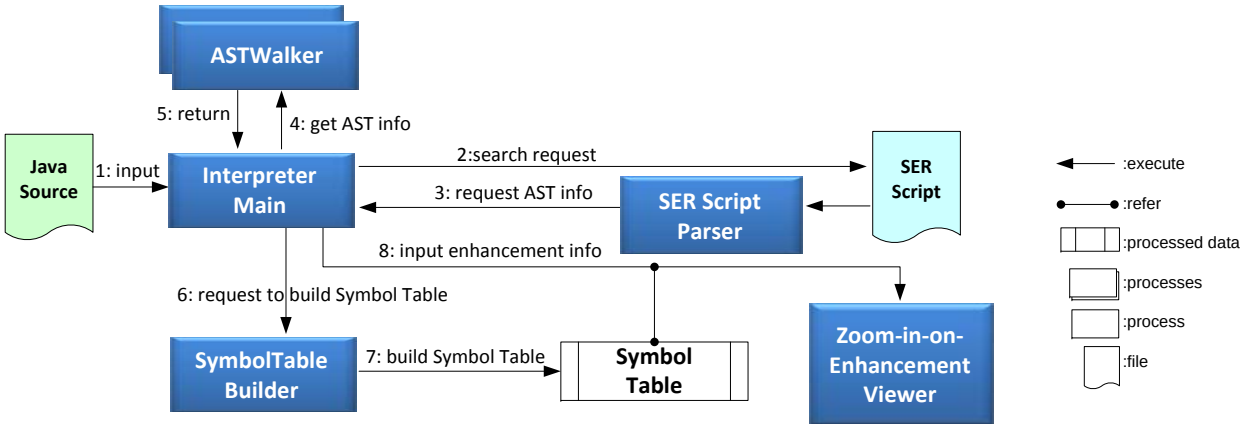

Figure 8.5: The zoom-in-on-enhancement view collaboration diagram.

document the enhancements used in the example applications. The views have been integrated with Eclipse.

Figure 8.5 presents a collaboration diagram that shows the backend processing triggered by the source editor to launch *a zoom-in-on-enhancements view*. When the SER interpreter's main module receives a Java source file as input, the corresponding SER script is identified (using a con-

figuration file), loaded, and parsed. The interpreter employs several abstract syntax tree walkers (using Visitors) to traverse the Java program, collecting the information about how the general enhancement instructions in the SER script will affect the specific program constructs (e.g., fields, methods, constructors, etc.). The collected enhancement information is stored in a symbol table for fast searching and retrieval. Finally, the interpreter compiles a complete documentation of the enhancements, which it uses to parameterize the zoom-in-on-enhancement viewer.

8.1.2 A Symbolic Debugger for Enhanced Intermediate Code

Because intermediate code enhancements are not represented at the source code level, source-level debugging of such enhanced bytecode is nontrivial. Application code is enhanced to be able to interact with a framework, and the enhancements cannot be simply turned off to facilitate debugging. Thus, tracing, analyzing, and fixing flawed programs whose bytecode has been enhanced with a standard debugger is misleading—the debugger will show all the enhanced program's code faithfully, both the original logic and the transparently introduced enhancements. From the debugging perspective, enhancements obfuscate the original source code's logic.

The debugging of transparently enhanced programs can be facilitated by making a symbolic debugger aware of the enhancements. The debugger could execute an enhanced program, but report the source code information pertaining to the original source code. As our proof of concept, we have created a new debugging architecture that leverages the facilities offered by the Java Platform Debugger Architecture (JPDA) [140]. We then applied the new architecture to augment the capabilities of the standard debugger distributed with Sun's JDK with an awareness of the enhancements in debugged programs.

Figure 8.6 demonstrates our new debugging architecture, which integrates a SER interpreter. When debugging enhanced bytecode, the debugger also takes the SER description of the enhancements as input. The integrated SER interpreter then computes *symbolic undo instructions* that map the enhanced bytecode to the original source code.

To reverse the enhancements that add and change program constructs, our debugging architecture introduces the *skip* and *reverse* symbolic undo instructions, which can be applied to classes, methods, fields, and entire packages. The debugger organizes the symbolic undo instructions as a hierarchical collection through the "contains" relationship (i.e., packages contain classes, classes contain methods, etc.). All the instructions are sorted in the order of their qualified full names,

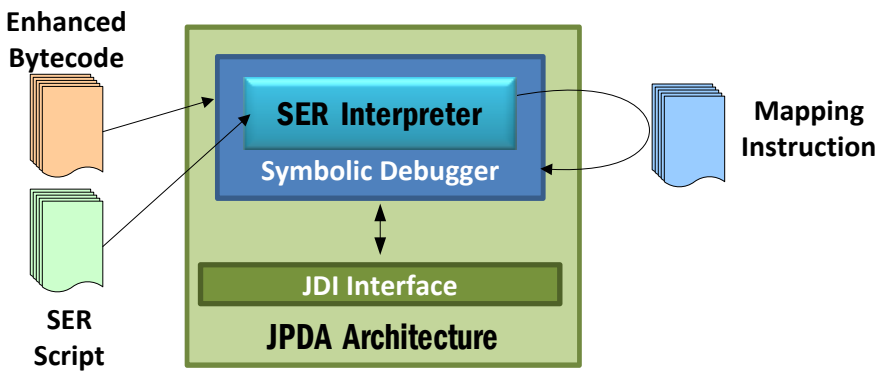

Figure 8.6: Debugging transparently enhanced programs.

so that they could be efficiently located through binary search in logarithmic time. The debugger executes the symbolic undo instructions on the encountered enhancements, so that the information reported to the user pertains to the original programmer written code.

The symbolic undo instructions work as follows. The *skip* instruction suppresses the output of those program elements that, in the original version, have not been represented in source code. Skip operations raise a special purpose debugging event whose semantics is similar to the regular debugger's *step* event; however, the output associated with handling the event is suppressed. The *reverse* instruction changes the name of a program construct displayed through the standard debugging output. For example, if an enhancer has changed the name of a class, a reverse instruction will direct the debugger to report the class's original name.

From the user's perspective, our symbolic debugger is a plug-in replacement for the standard JDK command-line debugger, providing the capabilities to step through the code, set breakpoints, print variable values, etc. The implementation leverages the Java Platform Debugger Architecture (JPDA)[140], which consists of several layers of protocols and interfaces provided by the Java Virtual Machine. The functionality required for symbolically undoing transparent enhancements is implemented by inserting additional translation logic to the standard operations used by debuggers based on JPDA. JPDA includes a client interface for accessing the debugging services, which are connected to the JVM through an event queue. To receive events from the queue, the client must set a breakpoint by calling an API method. Triggering a breakpoint delivers a debugging event to an event handler method that can access all the breakpoint's information, including its location, value of variables,etc.

Events are also triggered when the *step* command moves the debugger to the next source code

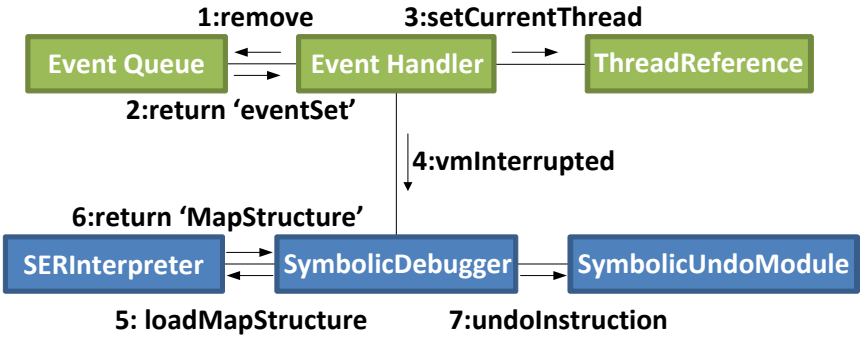

Figure 8.7: The symbolic debugger's collaboration diagram.

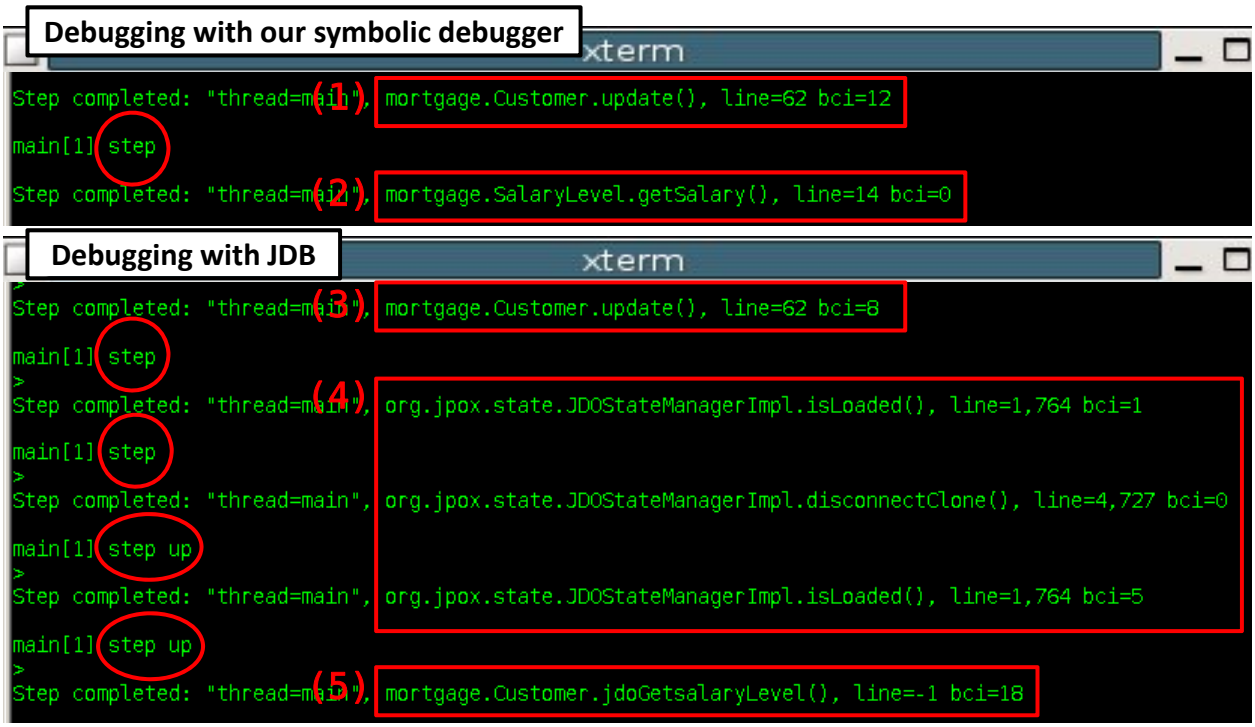

Figure 8.8: Debugging enhanced code: Debugger with an enhancements awareness vs. JDB

line. The values of member and local variables can be printed at any point after a breakpoint has been triggered or the step command has been executed. Our symbolic debugger intercepts each debugging event and executes a corresponding symbolic undo instruction, thus mapping the enhanced bytecode back to the original version of the code.

Figure 8.7 shows a collaboration diagram that details the runtime architecture of our symbolic

debugger. The diagram depicts the main events driving the execution of our symbolic debugger. The main module of the debugger, *SymbolicDebugger*, manages the symbolic undo information, using the services of the *EventHandler*. The *EventHandler* receives debugging events from the JPDA *EventQueue* and delegates them to *SymbolicDebugger* by calling `vmInterrupted`. *SymbolicDebugger* then evaluates the received event against the symbolic undo information, which can be generated on demand, and symbolically undoes any encountered enhancement. By using JPDA, which is thread-aware, our debugger can effectively handle multi-threaded programs.

As a demonstration of the utility of the new debugger, we inserted a bug to the example mortgage eligibility application presented in Section 3.1. We then traced the bug using both our augmented debugger and JDB. In our experiences, the enhancements introduced by the JDO framework complicate the debugging process. Figure 8.8 shows two screen shots, corresponding to debugging with our symbolic debugger and JDB, respectively. Points marked as (1) and (3) mark the start of the traced method, in their respective debugger's displays. Individual debugging steps are circled. Point (2) shows the original code as displayed by our debugger. Points (4) and (5) show the bytecode instructions that would be skipped and reversed by our debugger, respectively. Our experience suggests that our debugger has the potential to become an effective aid in locating bugs in enhanced programs. Nevertheless, only a controlled user study can confirm the veracity of this conjecture. We plan to conduct such a study as future work.

8.1.3 Discussion

We demonstrate how these transformations can be perused by the programmer in the zoom-in-on-enhancements view and can also enrich the functionality of bytecode-level programming tools in the symbolic debugger. Specifically, our SER interpreter automatically derives the transformations between the original and enhanced program versions and vice versa. Using a SER script, our interpreter calculates a precise mapping between a given piece of source code and its enhanced bytecode representation. This mapping can be effectively utilized by programming tools to improve their precision and utility.

The main contribution of this research is the SER language and its interpreter, whose expressiveness we validate by building an enhanced source editor and a symbolic debugger. The fact that we were able to build functional source level tools that work with four different enhancement strategies, in our view, validates the power of SER and the effectiveness of its interpreter. Although user studies with real programmers (as part of future work) are likely to reveal interesting insights,

giving us direct feedback from programmers, these user studies will only be tangentially related to the main contributions of this research.

SER scripts are the responsibility of framework developers, who will have to update the scripts to reflect the latest version of the enhancement strategy in place. This does require an extra maintenance effort, which we argue should be insignificant due to the declarative nature of SER.

Troubleshooting and fine-tuning programs often requires that the programmer know exactly which parts of the programs runtime representation come from programmer written code and which ones were introduced through bytecode enhancement. Existing programming tools fail to provide this information.

Our experiences with several commercial frameworks (JDO, Hibernate, JSecurity) and research projects indicate that bytecode enhancers avoid changing method bodies with the exception for replacing field accesses with setter/getter methods. Our choice of the structural changes to support is thus based only on empirical evidence. Our hypothesis is that changing bytecode on a larger scale safely requires expensive program analysis, which can be infeasible to perform at class load time without imposing a significant performance overhead. If bytecode enhancers start making larger-scale changes in the future, SER will have to be expanded accordingly.

8.2 Case Study: Reusable Enterprise Metadata with PBSE

To validate the advantages of PBSE as compared to XML and annotations, we have conducted a case study with two J2EE reference applications and a medium sized commercial application. As our subject applications, we used JPetStore [139] and JAdventureBuilder(JAB) [137]—well-known reference applications that demonstrate various Java enterprise technologies. As a larger-scale subject application, we used the Prescription Monitoring Program(PMP)[1]—a real-world enterprise application built according to the J2EE [143] three tier architecture.

All three applications included transparent persistence functionality, implemented with EJB 2. This version of the framework employs XML configuration files as its metadata format. For our case study, we first upgraded our subject applications to EJB 3, which is based on annotations. Then, based on the annotated versions, we produced a corresponding PBSE metadata specification. As it turned out, all three of our subject applications needed only a single PBSE metadata specification,

[1]Developed by T4G (http://www.t4g.com).

one written according to the structural source code patterns of the Java Persistence API (JPA) [138]. This specification, used in a prior example, appears in Figure 4.2 (2).

Table 8.1 compares the conciseness of each metadata format. As is expected, the total number of lines taken by both XML configuration files or annotations is directly proportional to the size of the application. Annotations manage to express the same metadata information significantly more concisely than XML (between 75% and 90% fewer lines of code on average). In addition to its reusability advantages, PBSE specifications are also quite concise. Compared to annotations, they express the same information in between 64% and 90% fewer lines of code on average. Of course, these numbers are dependent on the total size of the application, and PBSE leverages the fact that JPA metadata follows well-defined structural patterns.

Lines of code	JPetStore	JAB	PMP
Total source code	11,298	10,836	37,621
XML	503	538	890
Annotations	46	75	236
PBSE	17	17	17

Table 8.1: Conciseness of different metadata formats.

Reducing the amount of maintained hand-written source code offers tangible software engineering benefits. Since software complexity grows exponentially in relation to the size of a program [11], every line of programmer written code contributes to the software maintenance burden. Program changes to address new requirements or fix program defects require a program maintenance effort that is directly proportional to the size of a program [53].

Compared to both XML and annotations, PBSE provides programmability, understandability, maintenance, and reusability advantages, summarized next. PBSE also has certain applicability constraints that are discussed afterwards.

8.2.1 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 enterprise

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.

8.2.2 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. 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 invariants that hold throughout the entire codebase.

8.2.3 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.

8.2.4 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.

8.2.5 Discussion

The advantages of PBSE are due to the structural patterns between the source code and metadata of modern enterprise 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.

```
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
      ...
```

Figure 8.9: Identifying getters for `private` fields.

The reliance on the naming conventions imposed by enterprise frameworks makes PBSE susceptible to what is known in AOP as the *fragile pointcut problem* [79]. 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 naming conventions and programming discipline imposed by enterprise 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. 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"[2]. The PBSE examples introduced earlier would incorrectly capture such methods as getter. The PBSE declaration in Figure 8.9 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.

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

Although enterprise 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 [79, 134] and pointcut rejuvenation [69], 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. [1] 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 enterprise 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.

8.3 Case Study: Automated Cross-Language Metadata Translation

To validate this research, we applied it to reuse four additional concern 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 additional concern implementations in X10.

```
1  public class Integrate {
2      static val epsilon = 1.0e-9;
3      static def computeArea(left:double, right:double) {
4          return recEval(left, (left*left+1.0)*left, right, (right*right+1.0)*right, 0);
5      }
6      static def recEval(l:double, fl:double, r:double, fr:double, a:double) {
7          ...
8          finish {
9          async { expr1 = recEval(c, fc, r, fr, ar); };
10         expr2 = recEval(l, fl, c, fc, al);
11         }
12         return expr1 + expr2;
13     }
14 }
```

Figure 8.10: An X10 Integrate class to be unit tested.

8.3.1 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 additional concern, 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 additional concern in the literature, unit testing is indeed an additional concern. 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 8.10 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—$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{b-a}{2}x + \frac{b+a}{2})dx$.

Gaussian quadrature is non-trivial to implement correctly, but this implementation is even more

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

```
1  public class IntegrateTest {
2     var parm : double;
3     var expt : double;
4     var integrate : Integrate;
5
6     def init() {
7      integrate = new Integrate();
8     }
9
10    def finish() {
11     integrate = null;
12    }
13
14    def this(p : double, e : double) {
15     this.parm = p;
16     this.expt = e;
17    }
18    public def testComputeArea() {
19       val result = integrate.
20           computeArea(0, this.parm);
21       TUnit.assertEquals(expt, result);
22    }
23
24    public static def data() {
25       val parm = new Array[double
             ](0..1*0..2);
26       parm(0, 0) = 2;
27       parm(0, 1) = 6.000000262757339;
28       parm(1, 0) = 4;
29       parm(1, 1) = 72.000000629253464;
30       parm(2, 0) = 6;
31       parm(2, 1) = 342.000001284044629;
32       return parm;
33 }}
```

Figure 8.11: The unit testing class for the X10 Integrate class in Figure 8.10.

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 [56]). 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.

Our approach makes it possible to reuse the implementations of this additional concern. 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 8.10 is shown in Figure 8.11.

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 8.12, the programmer also has to declare a simple metadata specification shown in Figure 8.13. This specification establishes a coding convention as the one used in class `IntegrateTest`.

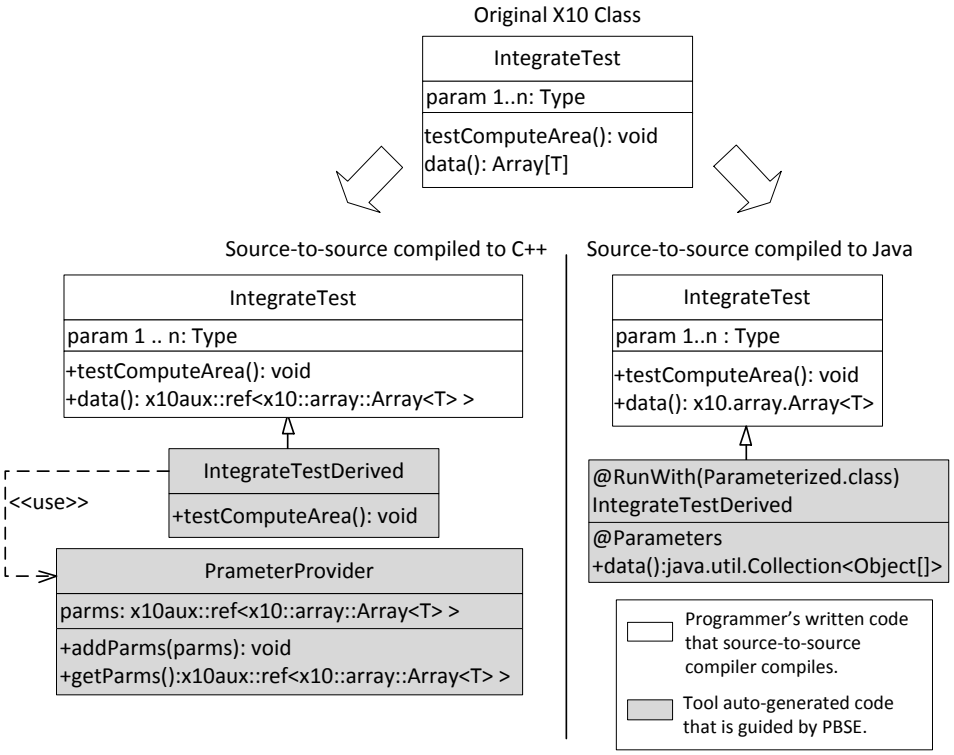

Figure 8.12: The class diagram for unit testing X10 programs with JUnit and CppUnit.

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, this research 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 this research is that it addresses the incongruity of features in different additional concern 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 8.12). In addition, JUnit requires that the method providing the parameters for unit test instantiations return `java.util.Collection` (the right part of Figure 8.12). Because `x10.array.Array`, the return type of the emitted Java method `data`, does not extend `java.util.Collection`, the auxiliary code generator uses the the Adaptor design pattern. To ensure that the `data` methods return the required `java.util. Collection`, an adaptor method wraps the returned type to an instance of `java.util.ArrayList`, thus satisfying this JUnit convention (Figure 8.14).

Supporting parameterized unit test execution in CppUnit requires more elaborate code generation.

```
Metadata UnitTest<Package p>
    Class c in p
    Where(public class *Test)
        c += @RunWith
        @RunWith.value = "Parameterized"
        TestMethod<c>
Metadata TestMethod<Class c>
    Method m in c
    Where (public def init ())
        m += @Before
    Where (public def finish ())
        m += @After
    Where (public def test* ())
        m += @Test
    Where (public static def data ())
        m += @Parameters
UnitTest<"integrate">
```

Figure 8.13: The PBSE specification for unit testing the X10 program.

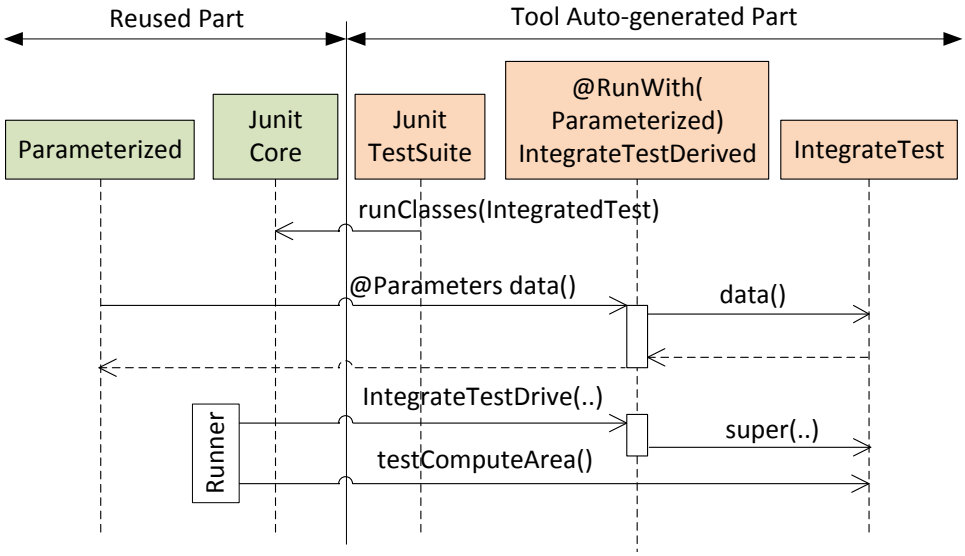

Figure 8.14: The sequence diagram for unit testing X10 programs with JUnit.

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

```
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 8.15: Extended macros based on CppUnit.

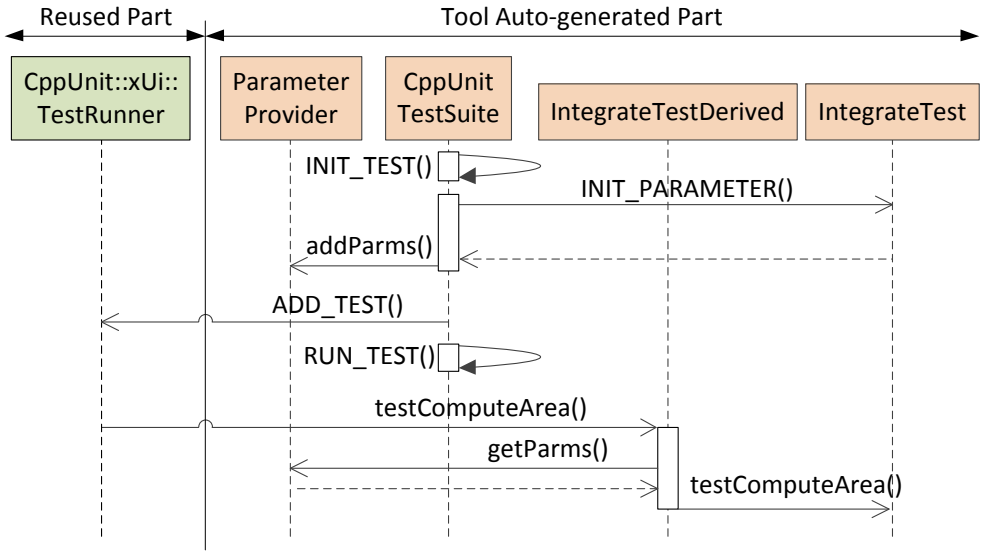

Figure 8.16: The sequence diagram for unit testing X10 programs with CppUnit.

The macros in Figure 8.15 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 8.16).

Standard implementations of additional concerns 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 sup-

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

```
public class Fmm3d{
    def getDirectEnergy() : Double{
    val model = new FmmModel();
    val directEnergy = finish (SumReducer()){
        ateach (p1 in locallyEssentialTrees) {
            var thisPlaceEnergy : Double = 0.0;
                for ([x1,y1,z1] in lowestLevelBoxes.dist(here)){
                    val box1 = lowestLevelBoxes(x1,y1,z1) as FmmLeafBox;
                    for ([atomIndex1] in 0..(box1.atoms.size()-1)){
                        for (p in uList){
                            for ([otherBoxAtomIndex] in 0..(boxAtoms.size-1)){
                                thisPlaceEnergy += atom1.charge*atom2Packed.charge/
                                atom1.centre.distance(atom2Packed.centre);
                            }
                        }
                    }
                    model.setModelId(id(box1.x,box1.y,box1.z));
                    model.setEnergy(thisPlaceEnergy);
                    // ...
                    TP.setFmmModelObj(model);
                }
                offer thisPlaceEnergy;
            }
        };
        return directEnergy;
    }
}
```

Figure 8.17: A persisting class `Fmm3d` (simplified version) for the X10 `FmmModel` class in Figure.

ported 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.

8.3.2 Transparently Persisting X10 Programs

Next we describe how we applied this research 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 8.17 shows an X10 class `Fmm3d` [96] that implements the Fast Multipole Method for electrostatic calculations with analytic expansions. The implementation is real and current: it follows the

```
<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 8.18: 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 8.19: PBSE for transparent persistence in Java.

strategy outlined by White and Head-Gordon [155] which was recently enhanced by Lashuk et al. [81]. 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.

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 5.2). Based on this specification, our approach in this research 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 additional concern 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 8.18 depicts a segment of the generated JDO XML deployment descriptor. To generate this deployment descriptor, our approach uses the PBSE depicted in Figure 8.19. Parameterized with this descriptor, the JDO runtime can transparently persist the specified X10 fields when the program is compiled to Java. Figure 8.20 depicts a segment of the generated ODB pragma definitions. To generate these pragmas, our approach uses the PBSE depicted in Figure 8.21. 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.

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.

```
#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 8.20: 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 8.21: PBSE for transparent persistence in C++.

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 [125]. 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.

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 8.22 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.

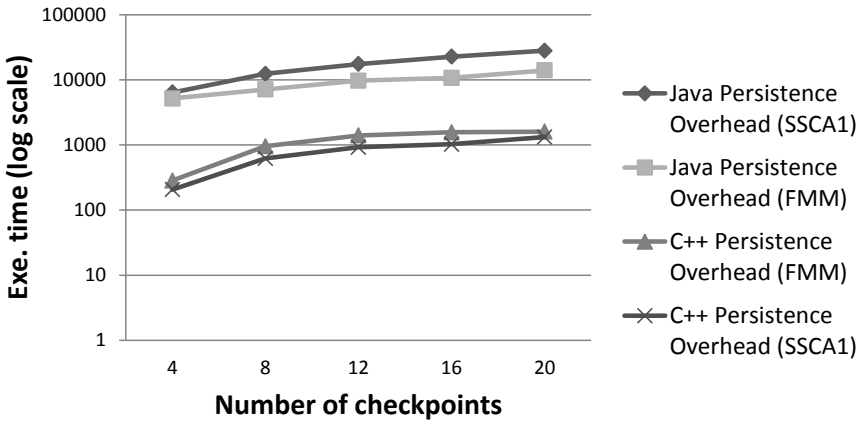

Figure 8.22: Persistence overhead for both backends.

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

8.3.3 Discussion

This research leverages the prevalence of declarative abstractions for expressing additional concerns 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 this research.

This research would be inapplicable if additional concerns were implemented through custom coding in mainstream languages. In fact, when reusing unit testing functionality, this research 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 additional concerns. 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.

8.4 Case Study: Checking and Inferring Metadata Invariants

Even though metadata invariants can still be useful if written by hand, inferring likely metadata invariants from large, established codebases saves development effort and time. To assess how effective our metadata invariant inference algorithm is, we have conducted case studies with seven open-source applications that rely on metadata.

The purpose of conducting the following case studies was to ensure that our algorithm can indeed infer likely program invariants that can later be refined by the programmer or used as is to maintain metadata consistency and correctness in the presence of program evolution. All the subject applications were large, open-source solutions for the enterprise domain: an Object Relational Mapping (ORM) system, an integrated development environment (IDE), a business process manager, and

a VPN server—Hibernate[6], JEdit[7], Spring framework[8], JBoss Seam framework[9], IntelliJ IDEA[10], RunaWFE[11] and OpenVPN ALS[12]. For each subject application, we ran our metadata inference algorithm with a threshold of 96%.

Table 8.2 presents the results. For each application, one metadata invariant was inferred and later verified through manual inspection. The right most column displays the inferred invariants in MIL. Some of these invariants can be checked in other unrelated applications that use the same framework, while others would need to be first refined and generalized by hand.

8.4.1 Inferring Metadata Invariants

The first invariant was inferred from the testing harness code of the Hibernate system. As it turns out, this system does not use any of the advanced JUnit features, as none of the **`public`** test suite methods were annotated with `@Before` or `@After`. No parameterized tests (i.e., annotated with `@Parameterized.Class`) were discovered either. The programmer who wants to use this inferred invariant on a more advanced usage example of JUnit would have to extend the automatically generated MIL invariant to look like the one that appears in Figure 6.3.

The second invariant was inferred from the popular JEdit editor. This invariant codifies the convention guiding the use of the built-in `@Override` annotation that marks overriding methods in subclasses. As it turned out, JEdit applies this annotation in over 96% of all cases, meaning that the remaining 4% constitute a coding inconsistency. This invariant can be applied as is to any Java application.

The third invariant was inferred from Spring, a widely used JEE framework. This invariant captures how well-written code tends to follow intuitive naming conventions. Even though the `@Configuration` annotations enables the programmer to name their configuration classes arbitrarily, the principles of self-documenting code still require intuitive program construct names. Checking this invariant can remind the programmer who creates a new configuration class (intuitively named) and forgets to annotate it with `Configuration`. Leaving out this annotation will cause the runtime system to ignore the new configuration class.

[6]Hibernate library – `http://www.hibernate.org/`
[7]JEdit text editor – `http://www.jedit.org/`
[8]Spring framework – `http://www.springsource.org/`
[9]JBoss Seam framework – `http://www.seamframework.org`
[10]IntelliJ IDEA – `http://www.jetbrains.com/idea/`
[11]Enterprise business process manager – `http://wf.runa.ru/`
[12]OpenVPN ALS server – `http://openvpn-als.sourceforge.net/`

Metadata Type	Application	Application Size (Files / LOC)	Metadata Size	Inferencing time (ms)	Inferred Metadata Invariant in MIL

Annotations	Hibernate core 4.0.0	3,957 / 285,653	2243	18,754	`Class c in p`
					`Where (* class *Test | * class Test*)`
					`  Method m in c`
					`  Where (public void *)`
					`  Assert (@Test m)`

	JEdit 4.3.3	531 / 109,548	106	18,109	`Class c in p`
					`Method m in c`
					`Where (@Override m)`
					`  Assert (c.super has m)`

	Spring 3.1.0.M2	4,465 / 353,769	66	14,737	`Class c in p`
					`Where (* class *Configuration | * class *Config)`
					`  Assert (@Configuration c)`

	JBoss Seam 3.0.0	1,228 / 69,505	97	13,899	`Class c in p`
					`Where (@XmlType* class org.jboss.seam*)`
					`  Field f in c`
					`  Where (f)`
					`  AssertExists (@XmlType.propOrder has f.name)`

	IntelliJ 10.5.0	27,901 / 1,913,330	62	154,060	`Field f in c`
					`Where (@Attribute f)`
					`  Assert ((f.name eq @Attribute.name)|`
					`  (*f.name eq @Attribute.name)|`
					`  (*Lc(f.name)eq @Attribute.name))`

XML	RunaWFE 3.4.0	1,585 / 111,012	90	3,306	`Class c in p`
					`Where (* class * not "org.jbpm.identity.*")`
					`  Assert (<table>.name eq ("JBPM_"* + Uc(c.name))`

	OpenVPN ALS 0.9	1,954 / 165,000	93	2,446	`Class c in p`
					`Where (public class *)`
					`  Assert ((<form-bean>.name eq Lc(c.name))|`
					`  (*<form-bean>.name eq c.name))`

Table 8.2: Inferring Metadata Invariants from Third-Party Applications.

The fourth invariant was inferred from Seam, a JBoss framework for constructing web-applications. This framework uses both annotations and XML metadata. To bind Java class fields to XML names, the `propOrder` array attribute of the `@XmlType` annotation contains the names of all the fields. At runtime, these fields are bound to their corresponding values in the XML file. This metadata invariant fills the unique niche of checking this programming convention, whose violation leads to obscure runtime errors. The `AssertExists` MIL construct ensures that the names of all class fields appear as string values of the `propOrder` array (in any order). If, say, the programmer adds a new field to the class, but forgets to simultaneously add its name to the `propOrder` array, the metadata invariants checker will promptly alert the programmer, thereby avoiding a difficult-to-trace runtime error.

The fifth invariant was inferred from the popular IntelliJ Java IDE. The invariant expresses a Java format representation of an XML document being mapped to the actual document. In IntelliJ, The `@Attribute` annotation happens to form a naming relationship with the tagged field's name. They either match exactly, or the annotation's name attribute matches the field's suffix exactly or case-insensitively. There is value in keeping the names in the main source code and its XML representation consistent to facilitate both program comprehension and maintenance. Thus, checking this invariant can help uncover some naming inconsistencies that are likely to incur an unnecessary maintenance burden.

The sixth invariant was inferred from RunaWFE, an enterprise business process manager that integrates the JBoss-jBPM workflow core to bridge business analysts and developers. In addition to Java 5 annotations, RunaWFE uses XML configuration files. The `not` operator excludes the package for which the invariant does not hold. The inferred invariant codifies the naming relationship between the name attribute of the XML node `<table>` and the name of the bound class. This convention is common in transparent persistence code configured through XML. Checking this invariant statically can help ensure that all the classes are properly bound, and no runtime errors will occur due to misnaming Java class names in the XML descriptor.

The seventh invariant was inferred from OpenVPN ALS, a web-based SSL VPN server written in Java. Apache Struts provides standard Java Web application functionality whose XML configuration files form an invariant codifying the relationship between the `name` attribute of the `<form-bean>` XML node and the bound class's name. As in the previous subject application, checking this invariant statically is likely to prevent mistypings and other inconsistencies from causing runtime errors.

8.4.2 Checking Metadata Invariants

As a practical implementation of metadata invariants, we have integrated our metadata invariant inferencer and checker with Eclipse IDE by means of its plug-in architecture. Specifically, our metadata invariants plug-in provides a graphical interface to our backend inference engine. The plug-in makes it possible to run the inferencer on the current project's source files and examine the generated MIL specifications. The inference portion of this research benefits from the IDE integration only superficially—the inferencer can be invoked from the command line or as part of a build script with the same results.

The component that benefits the most from Eclipse integration is the metadata invariants checker, which is run every time the programmer saves a source file. The invariants checker is parameterized with a MIL input file that contains a list of metadata invariants that should be maintained for a given project. After the programmer modifies a source code file, either by enhancing it with new functionality or improving the code through a refactoring, our metadata refactoring runs and displays the violated metadata invariants in the error window. Upon examining the violated invariants, the programmer is then free to take corrective actions. For example, the programmer may edit a metadata specification to keep it in sync with the latest source code change. Alternatively, the programmer may undo a refactoring if the violated metadata invariant is too burdensome to fix. By reporting the violated metadata invariants, this research provides the programmer with the knowledge about how the latest step in evolving the code affects its correctness with respect to metadata. As with the majority of bug finding tools, it is the programmer's responsibility to confirm the reported suspected bugs and fix them if necessary.

8.4.3 Discussion

Table 8.2 shows how this research can infer metadata invariants from third-party applications that use either annotations or XML as their metadata format. These case studies have shown that the metadata invariants found in these applications mostly codify some implicit (undocumented) programming conventions. Inferring metadata automatically is a facility that can help the programmer. Nevertheless, even without inferring the invariants, checking manually composed metadata invariants is still beneficial. The programmer can use MIL to write metadata invariants from scratch or to refine those inferred invariants that lack the desired accuracy.

Concerns	Annotations	APC in JS	# Advice (LOC)	# Aspect	# PC
Persistence	@Persist (key = "key_variable", value = "variable name")	Variable	2 (6)	2	2
Security	@Security (kind = Security.op.Encrypt, variable = "variable name")	Variable	1 (3)	1	1
	@Security (kind = Security.op.Decrypt, variable = "variable name")	Variable	1 (3)	1	1
Transactions	@Transaction (inspect = "inspect_function", function = "function name")	Function	1 (6)	1	1

Table 8.3: The case studies metrics (i.e., persistence, security, and transactions). (APC in JS: Annotated Program Constructs in JavaScript.)

In some cases, this research infers irrelevant invariants that should be discarded. For example, in IntelliJ, we discarded four irrelevant invariants that were not applicable to close to 10% of the codebase. Furthermore, in OpenVPN, we discarded the following false positive: `Assert(*Uc(<form-bean>.name)eq c.name)`. Our inference algorithm is unfortunately sensitive to any significant deviation from established naming convention. These deviations can be identified and removed from the input given to our algorithm, thereby improving the algorithm's precision and effectiveness.

8.5 Case Study: Enhancing JavaScript Programs via Java Annotations

For our case studies, we replicated the functionalities provided by three third-party libraries that enhance JavaScript programs with persistence [22], security [133], and transactions [33].

Although we found the design and implementation of these third-party libraries compelling, the thesis of this work is that concerns can be added to JavaScript code declaratively, and that automated program transformation can eliminate the need to manually modify the maintained version of the source code. Hence, the purpose of our case studies was to measure the programming effort incurred by the TAE-JS approach. The application developer's effort of annotating JavaScript

constructs is quite minimal. Therefore, our measurements aim at understanding the concern developer's effort, required to build a TAE-JS APT plug-in. Recall that the TAE-JS generates Java identifiers automatically irrespective of how the application developer annotates them.

For each of the libraries, we next first briefly describe the library's functionality and programming interface. Then, we explain how we replicated the same functionality declaratively via annotations and describe the TAE-JS plug-in to introduce the library calls into unaware JavaScript programs. To ensure that we have managed to replicate the original functionality faithfully, we tested the TAE-JS approach for each added concern on a Web application running in a browser.

8.5.1 Persistence for JavaScript

Although modern Web browsers can persist client-side data for offline use, JavaScript developers may find it difficult to manipulate and synchronize the persistence state saved in server-side remote storage. To address these issues, Cannon and Wohlstadter [22] introduced a persistence framework for JavaScript that offers persistence facilities similar to those found in frameworks for the Java language (e.g., Hibernate and JDO[13]). Specifically, the introduced framework can detect mutations of persistent objects, serialize persistent objects to store them locally, and synchronize local and remote copies of persistent objects. The persisted objects are represented as key/value pairs[14] and manipulated with explicit library calls.

As it turns out, the functionality of this persistence library lends itself well to being expressed declaratively. In TAE-JS, we annotate the persistent values with the `@Persist` annotation that takes two String attributes: `key` and `value`. For example, to persist variable `foo`, the programmer can annotate it as `@Persist (key= "key_foo", value= "foo")`.

To generate an AspectScript aspect to weave in the persistence library calls into a program, the generator also needs the information about the scope of the persisted variables. In JavaScript, variables can be local, member, and global. AspectScript can interpose advise when accessing or modifying all these variable types, albeit through different pointcuts. The annotation editor communicates this information through generated Java identifiers. For local and member variables, the name of the enclosing functions are represented as a Java class. For global variables, the name of the class is `GLOBAL`—e.g., `GLOBAL gvar;`. Based on this information, the APT persistence plug-in intercepts accessing and modifying variables by means of `around` and `after` advice mechanisms, respectively.

[13] JDO – `http://www.datanucleus.org/products/accessplatform_3_0/jdo/`
[14] Web Storage – `http://www.w3.org/TR/2009/WD-webstorage-20091029/`

The advice functions simply contain the library calls to store and retrieve the persistent variables' values from persistent storage. As shown in the first row of Table 8.3, the programmer can render a variable persistent just by annotating it, with TAE-JS automatically generating two advices, two aspects, and two pointcut expressions.

Rendering Yahoo! Finance E-Chart Persistent

To evaluate how well the TAE-JS approach can scale, we used it to render *all* the variables in the initialization functions in the `echart_head.js` script from the Yahoo! Finance website[15] persistent. That is, every time this page is reloaded, its variables are initialized to the values they held the last time the page was displayed. Although one cannot make a compelling business case for persisting all the variables, we conducted this study to test the scalability of the TAE-JS code generation infrastructure. In terms of the specific numbers involved, there were 167 variables tagged with the `@Persist` annotation. The TAE-JS IDE plug-in generated 53 Java classes for each JavaScript function that contained variables. All the variables were added to class `PersistenceAnnotator`, which was passed as a parameter to the persistence APT plug-in. The plug-in generated 3,192 lines of AspectScript code required for adding the persistence library calls for each variable read and write. Although one can hardly imagine a scenario under which so many variables would have to be rendered persistent, the TAE-JS code generation infrastructure was able to generate the required aspect code for this input almost instantaneously.

8.5.2 Security for JavaScript

To protect sensitive information, Web applications may need to encrypt some JavaScript variables before transmitting them to the server and decrypt the encrypted values received from the server. To that end, Stark et. al. [133] presented a JavaScript symmetric encryption library, whose implementation is specifically optimized for JavaScript. This general-purpose encryption library provides a simple API that the JavaScript programmer can use to encrypt and decrypt variables.

With TAE-JS, the functionality of this library is exposed through the `@Security` annotation that has two attributes: the `kind` of security operation performed, and the `variable` operated on. The first attribute is a Java `enum` type, which can be typechecked more precisely than a string attribute. Also, this annotation is easily extensible. One can add a new encryption mechanism by creating

[15] Yahoo Finance – `http://finance.yahoo.com/`

```
var pcEncryptV1 = AspectScript.Pointcuts.get("pd");
var pcDecryptV1 = AspectScript.Pointcuts.set("mid");

var adviceEncryptV1 = function(jp) {
    return encrypt(jp.value);
};

var adviceDecryptV1 = function(jp) {
    return decrypt(jp.value);
};

var aspectEncryptV1 = AspectScript.aspect(AspectScript.AROUND, pcEncryptV1,
    adviceEncryptV1);
AspectScript.deployOn(aspectEncryptV1, checkCredentials);

var aspectDecryptV1 = AspectScript.aspect(AspectScript.AROUND, pcDecryptV1,
    adviceDecryptV1);
AspectScript.deployOn(aspectDecryptV1, getMemberId);
```

Figure 8.23: AspectScript code enhanced with security.

```
function sample () {
    // Code_Block_A: upper code block of transaction .
    // Code_Block_T: transactions code block to be selected .
    // Code_Block_B: bottom code block of transaction .
}
```

Figure 8.24: Original code snippets.

another `enum` constant. To encrypt a variable `foo`, the programmer annotates it as `@Security( kind= Security.op.Encrypt, variable= "foo")`.

The strategy for generating AspectScript code to express this concern is similar to that used for the persistence concern. The similarity stems from the fact that both of these concerns are applied to variables within a given function. Figure 8.23 shows a snippet of AspectScript code that encrypts variable `pd` in function `checkCredentials` on the client side, as well as decrypts variable `mid` in function `getMemberId`.

As shown in the second and third rows of Table 8.3, the programmer can encrypt or decrypt a variable by annotating it, with TAE-JS automatically generating 2 advices, 2 aspects, and 2 pointcut expressions for each annotation.

```
function sample () {
    // Code_Block_A
    sample_T(args);
    // Code_Block_B
}
```

```
function sample_T(args) {
    var tx = transaction {
    // Code_Block_T
    };
}
```

Figure 8.25: *Method Refactoring* from Figure 8.24.

```
var pcTranV1 = AspectScript.Pointcuts.exec(sample_T);

var adviceTranV1 = function(jp) {
    var tx = jp.args[0];
    if(inspect(tx))
        tx.commit();
};

AspectScript.after(pcTranV1, adviceTranV1);
```

Figure 8.26: AspectScript code enhanced with transactions.

8.5.3 Transactions for JavaScript

A common approach to improving security and reliability in the presence of untrusted third-party code is to execute that code in a transactional context. A unit of code delineated by a transaction is executed speculatively, and depending on the observed behavior, the results can be either committed or rolled back. When a transaction is rolled back, the program's state is restored to the point right before the transactional code started execution. To avail this powerful mechanism to Web applications, Dhawan et. al. [33] added transactions to JavaScript. As their implementation strategy, they extended the language with a new keyword, `transaction` that the programmer can use to delineate transaction boundaries; the implementation also includes a library, called Transcript, with the API for managing transactions.

TAE-JS enables the programmer to engage the services of the Transcript library declaratively, with a single annotation rendering a block of JavaScript code transactional. When a programmer selects a block of JavaScript code, our annotation-aware IDE warns the programmer that only variables and functions can be annotated, and then prompts the programmer if an automated Extract Function refactoring [46] should be performed. If the programmer agrees, the annotation editor opens to accept the TAE-JS transaction annotation. Then the IDE, behind the scenes, extracts the function to be executed transactionally, so that the subsequently generated AspectScript code could operate on the extracted function.

Figure 8.24 shows function `sample`, in which the programmer selects `Code_Block_T` to be rendered transactional. Because aspect languages cannot operate on arbitrary code blocks, the IDE offers to refactor the function, extracting function `sample_T` as shown in Figure 8.25. The calls to the Transcript library as shown in Figure 8.26 are then inserted to the extracted transactional function. Because AspectScript would not work on JavaScript extended with a new keyword (`transaction`), we tested the reference implementation using regular extracted JavaScript functions. However, if the `transaction` keyword is to be added to JavaScript, AspectScript will probably be extended with `transaction`-specific pointcuts.

8.5.4 Discussion

The ability to access the functionality written in another language goes all the way back to Common Lisp with its foreign function interface (FFI) [14]. Usually, FFI serves as a mechanism for improving performance by calling well-optimized routines written in another language or for accessing those legacy code parts that cannot be easily ported. To the best of our knowledge, TAE-JS is the first approach that enables a host language to reuse the declarative metadata facilities of another language. In other words, the motivation for using the functionality of a different language is to leverage the expressiveness of its metadata facility. Furthermore, TAE-JS makes it possible for Java to fully use the annotation facilities of JavaScript without extending the JavaScript syntax. Instead an IDE enables a multi-lingual development model, with the Java compiler ensuring proper name and typychecking of the entered annotations. Although a built-in metadata facility makes a programming language amenable to declarative programming models, it is not always feasible to add this facility to a widely used language with a large legacy codebase. Hence, leveraging the built-in metadata facility of another language presents a viable alternative. Next we discuss what we consider as the main advantages and limitations of the TAE-JS approach.

The main advantage of the TAE-JS approach is that it cleanly separates concerns. It can enhance the core functionality with additional concerns based on a declarative specification. The power of Java typechecking ensures that these specifications are syntactically correct. Furthermore, because TAE-JS encodes the information about the JavaScript constructs interacting with the added concerns as Java identifiers, APT plug-ins generating AspectScript code do not need to reference the JavaScript code. Finally, which concerns are to be added for a given deployment is configured entirely through build configuration.

One of the limitations of TAE-JS is that it adds concerns statically. As a result, the concerns would not appear in those parts of the code that are generated dynamically at runtime. In particular, JavaScript features the `eval` function that can evaluate a textual string at runtime, generating new JavaScript code. Assume that a field in a JavaScript function was annotated as persistent, and the same function contains an `eval` that generates code referencing the persistent field. The static transformations that render the field persistent would not be applied to the code generated by `eval` at runtime. We plan to address this limitation as a future work by offering a mechanism that can transform dynamically generated code.

TAE-JS generates aspects, an approach that presents two limitations. First, aspect languages cannot add functionality to arbitrary blocks of code that cannot be easily extracted into functions. As a result, only those concerns that are focused around variables and functions are amenable to be added via TAE-JS. The second limitation stems from AspectScript automatically transforming JavaScript programs to weave in concern code into the main code. Transformed code is hard to debug. Even though AspectScript does not yet feature a debugger, this problem has been addressed in aspect extensions for other languages. For example, AspectJ comes with a state-of-the-art symbolic debugger, and it is likely that a similar debugger will be provided for AspectScript.

Chapter 9

Conclusions and Future Work

This dissertation have explored some of the key issues of metadata-driven software development, with a particular emphasis on effective reuse and safe evolution. As software development paradigm has become highly dependent on the declarative model for realizing the majority of concerns, metadata has become an integral role of the modern computing application development. Programmers spend an increasing amount of time and efforts maintaining applications in the presence of metadata. As a result, enabling effective reuse and safe evolution in metadata-driven applications is likely to yield tangible software engineering benefits, improving software quality and increasing productivity. This dissertation has examined why metadata-driven software development suffers from poor reusability and unsafe evolution. Through innovation in domain-specific languages and automated code generation, the research described herein promotes disciplined and systematic software engineering approaches in metadata-driven software development.

9.1 Summary of Contributions

The research presented in this dissertations was published in the proceedings of OOPSLA'09 [128], AOSD'10 [148], AOSD'12 [131], and ICSE'12 [130]. The research prototypes described in this dissertation were presented as formal software demonstrations at OOPSLA'09 [127] and ICSE'12 [129]. Major contributions of this research include:

1. A novel approach [128, 127] to improving the precision and utility of source-level programming tools in the presence of intermediate code enhancements by concisely expressing structural enhancements with the Structural Enhancement Rule (SER) language, a Domain-Specific Language,

2. Pattern-Based Structural Expressions (PBSE) [148]—a new metadata format and its automated translation infrastructure that offers usability, reuse, and ease-of-evolution advantages addressing shortcomings of mainstream enterprise metadata formats, as compared to both XML and Java 5 annotations,

3. An approach [131] to reusing additional concern implementations of a mainstream language from an emerging language program by means of automated cross-language metadata translation, when the emerging language is compiled to the mainstream language,

4. *Metadata Invariants* [130]—a new abstraction for expressing the interconnections between the metadata and the source code of a program, and a practical algorithm for inferring likely metadata invariants than can be expressed in Metadata Invariants Language (MIL)—a domain-specific language (DSL), and

5. An approach for using the metadata infrastructure of another language in a host language, without extending the host language's syntax, in order to transparently enhancing JavaScript programs by means of generative aspects.

9.2 Future Work

The algorithms, techniques, and tools for supporting reuse and evolution in metadata-driven software development, explored by this dissertation, provide rich possibilities for future work. Each software technology developed for this research can be further improved for the perspective of its functionalities and applicability. Additionally, the general idea can be discussed for its applicability to other domains. We next outline several possible future work directions.

9.2.1 Improving the Awareness of Transparent Program Transformation

As future work, we plan to conduct user studies to evaluate the value of our approach for programmers with different levels of expertise. One such study could evaluate whether a symbolic undo debugger is more effective than a regular debugger in helping the programmer to locate and fix bugs. Another study could evaluate the value of integrating the enhancement information with a programming editor. We plan to create a debugger for SER scripts. Although the declarative nature of SER scripts makes it easier to ensure their correctness, bugs still can be introduced, particularly if a SER script is developed by someone other than a framework developer. Having a SER debugger is likely to improve the usability of our approach. We plan to extend our approach to programs that have been transparently enhanced more than once, possibly by different enhancers. For example, the same application can use multiple frameworks, each enhancing intermediate code

in its own way. Bytecode enhancement could add functionality in languages other than the host language, including query languages such as SQL or Datalog.

Our approach would have to be extended to add the enhancements-awareness to programming tools handling multi-language applications. There could be potential benefit in synthesizing the source code representation of bytecode-only enhancements. To that end, the SER interpreter would have to be integrated with a decompiler that is enhancements-aware. The success of this approach would mainly depend on the decompiler's efficiency and precision. Generating SER scripts automatically will reduce the framework programmer's effort and make our methodology more appealing to the average programmer. A promising approach would require generalizing program differencing, a target of several recent research efforts [75]. Finally, we would like to make our symbolic debugger available as an Eclipse IDE plug-in.

9.2.2 Increasing the Applicability of PBSE

The applicability of PBSE is not limited to Java programs only. It can be applied to other languages with built-in metadata facilities such as C# and its attributes. What is more interesting is to apply PBSE to older languages such as C and C++ as an alternative to XML configuration files. PBSE can be used not only for frameworks, but as input for automatic code generators and adapters. We plan to release our Eclipse Plug-in that automatically annotates Java code given a PBSE specification. The success of a new metadata format can only be ensured through a grassroots movement, with real enterprise developers trying out the new format and experiencing its benefits first hand.

9.2.3 Eliminating Interferences and Increasing Expressiveness of PBSE in the Presence of Multiple Concerns

When applied to the same codebase, additional concern implementations may harmfully interfere with each other. Although our approach does not change how additional concerns are implemented, but only how they are expressed, we plan to explore whether PBSE be extended with constructs that specify the order in which additional concerns should be applied. When multiple additional concerns 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 additional concerns. However, if portions of core functionality become expressible declaratively, the potential benefits of our ap-

proach will also increase. If metadata can be used to express certain core functionalities, metadata translation can supplement or, in some instances, replace compilation.

9.2.4 Extending the Applicability of Metadata Invariants

We introduced metadata invariants as a mechanism that can find bugs in metadata represented as XML or Java 5 annotations. We plan to extend our approach to validate the correctness in other metadata formats, including C/C++ pragmas and C# attributes. Other metadata formats similarly form relationships with the programs written in a mainstream programming language, a property that can be leveraged to verify their correctness. We plan to investigate whether our metadata invariant checker can be integrated with static bug finding tools such as FindBugs [62].

9.2.5 Enhancing Scalability and Expressiveness of TAE-JS

One future work direction will evaluate the scalability of the TAE-JS approach for large JavaScript codebases. Another direction will continue investigating the expressiveness of TAE-JS to enhance JavaScript code with concerns provided by other libraries. Finally, we plan to investigate how TAE-JS can be applied to dynamically generated JavaScript code.

Bibliography

[1] A. Abdelmeged, T. Skotiniotis, and K. J. Lieberherr. Controlled evolution of adaptive programs. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 89–98, New York, NY, USA, 2009. ACM.

[2] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, 2006.

[3] Apache Jakarta Project. The Byte Code Engineering Library. `http://jakarta.apache.org/bcel/manual.html`.

[4] T. Apiwattanapong, A. Orso, and M. Harrold. A differencing algorithm for object-oriented programs. *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 2–13, Sept. 2004.

[5] M. P. Atkinson. The Napier88 persistent programming language and environment, 1988.

[6] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[7] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1:355–398, 1992.

[8] C. Bauer, G. King, and I. NetLibrary. *Hibernate in Action*. Manning, 2005.

[9] D. M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, TCLTK*, 1996.

[10] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional programming (ICFP 2003)*, pages 51–63, 2003.

[11] D. Berry. *Academic Legitimacy of the Software Engineering Discipline*. Carnegie-Mellon University, Software Engineering Institute, 1992.

[12] C. Beust and H. Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.

[13] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. W3C, XQuery 1.0: An XML Query Language, 2007.

[14] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common Lisp object system specification. *ACM Sigplan Notices*, 23:1–142, 1988.

[15] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 365–383, New York, NY, USA, 2004. ACM.

[16] G. Brooks, G. J. Hansen, and S. Simmons. A new approach to debugging optimized code. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and Implementation*, pages 1–11, 1992.

[17] W. Brown, R. Malveau, H. McCormick III, and T. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc. New York, NY, USA, 1998.

[18] J. Bubba, A. Kaplan, and J. Wileden. The Exu approach to safe, transparent and lightweight interoperability. In *Proceedings of the 25th International Conference on Computer Software and Applications Conference, COMPSAC*, pages 393–400, 2001.

[19] O. Burn. Checkstyle. `http://checkstyle.sourceforge.net/`.

[20] B. Burton, R. Aragon, S. Bailey, K. Koehler, and L. Mayes. The Reusable Software Library. *Software, IEEE*, 4:25–33, 1987.

[21] G. Caldiera and V. R. Basili. Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24:61–70, 1991.

[22] B. Cannon and E. Wohlstadter. Automated object persistence for JavaScript. In *Proceedings of the 19th International Conference on World Wide Web, WWW*, pages 191–200, 2010.

[23] V. Cepa and M. Mezini. Declaring and enforcing dependencies between .NET custom attributes. In *Generative Programming and Component Engineering*, pages 319–331. 2004.

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

[25] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. *SIGPLAN Not.*, 34(5):13–24, 1999.

[26] T. Cleenewerck, K. Czarnecki, J. Striegnitz, and M. Völter. Evolution and Reuse of Language Specifications for DSLs (ERLS). In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP Workshops 2004)*, volume 3344. Springer, 2004.

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

[28] M. Copperman. Debugging optimized code without being misled. *ACM Trans. Program. Lang. Syst.*, 16(3):387–427, 1994.

[29] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.

[30] K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *In MoDELS*, pages 692–706, 2006.

[31] M. Dahm. Byte code engineering. *Java Informations Tage*, pages 267–277, 1999.

[32] T. Devadithya, K. Chiu, and W. Lu. C++ reflection for high performance problem solving environments. In *Proceedings of the 2007 spring simulation multiconference - Volume 2*, SpringSim '07, pages 435–440, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[33] M. Dhawan, C. chieh Shan, and V. Ganapathy. Enhancing JavaScript with Transactions. In *Proceedings of 26th European Conference Object-Oriented Programming, ECOOP*, pages 383–408, 2012.

[34] M. Dmitriev. Language-specific make technology for the Java programming language. *SIGPLAN Not.*, 37(11):373–385, 2002.

[35] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 118–128, New York, NY, USA, 2007. ACM.

[36] D. Duggan. A Mixin-Based, Semantics-Based Approach to Reusing Domain-Specific Programming Languages. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 179–200. Springer-Verlag, 2000.

[37] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, pages 207–239, 2006.

[38] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Software Composition*, pages 200–215. 2007.

[39] Eclipse Foundation. Eclipse Java development tools, March 2008. `http://www.eclipse.org/jdt`.

[40] M. Eichberg, T. Schäfer, and M. Mezini. Using annotations to check structural properties of classes. In *8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442, pages 237–252. Springer, 2005.

[41] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 275–284, New York, NY, USA, 2006. ACM.

[42] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 275–284, New York, NY, USA, 2006. ACM.

[43] R. E. Faith. *Debugging programs after structure-changing transformation*. PhD thesis, 1998. Adviser-Jan F. Prins.

[44] H. Fernau. Algorithms for learning regular expressions. In *Algorithmic Learning Theory*, pages 297–311. Springer, 2005.

[45] K. Fisher and R. Gruber. Pads: a domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 295–304, New York, NY, USA, 2005. ACM.

[46] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[47] J. Fox and R. Andersen. Using the R statistical computing environment to teach social statistics courses. 2005.

[48] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 31:529–536, 2005.

[49] P. Fritzson. A systematic approach to advanced debugging through incremental compilation (preliminary draft). In *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on High-level debugging*, pages 130–139, 1983.

[50] Glen McCluskey. Using Java Reflection. `http://java.sun.com/developer/technicalArticles/ALT/Reflection/index.html`.

[51] C. J. Godby, D. Smith, and E. Childress. Two paths to interoperable metadata. In *DCMI '03: Proceedings of the 2003 international conference on Dublin Core and metadata applications*, pages 1–9. Dublin Core Metadata Initiative, 2003.

[52] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[53] L. Gremillion. Determinants of program repair maintenance requirements. *Communications of the ACM*, 27(8):826–832, 1984.

[54] M. L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, 32:548–566, 1993.

[55] S. Haefliger, G. von Krogh, and S. Spaeth. Code Reuse in Open Source Software. *Management Science*, 54:180–193, 2008.

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

[57] J. Hennessy. Symbolic debugging of optimized code. *ACM Trans. Program. Lang. Syst.*, 4(3):323–344, 1982.

[58] M. A. Hernández, P. Papotti, and W.-C. Tan. Data exchange with data-metadata translations. *Proc. VLDB Endow.*, 1(1):260–273, 2008.

[59] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Not.*, 27(7):32–43, 1992.

[60] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages (POPL 2001)*, pages 67–80. ACM, 2001.

[61] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3:117–148, 2003.

[62] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, 2004.

[63] S. S. Huang and Y. Smaragdakis. Class morphing: Expressive and safe static reflection. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 79–89. ACM, June 2008.

[64] S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 399–424. Springer-Verlag, Aug. 2007.

[65] InfoEther, Inc. PMD. `http://pmd.sourceforge.net/`.

[66] T. Ishio, S. Kusumoto, and K. Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 178–187, 2004.

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

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

[69] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 09)*, 2009.

[70] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of 21th ACM SIGOPS Symposium on Operating Systems Principles, SOSP*, pages 17–30, 2007.

[71] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, London, UK, 2001. Springer-Verlag.

[72] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*. Springer-Verlag, 2001.

[73] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwing. Aspect-oriented programming. In *ECOOP*. Springer-Verlag, 1997.

[74] J. Kienzle and R. Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 37–61, London, UK, 2002.

[75] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, 2007.

[76] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[77] K. E. Kline and D. Kline. *SQL in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[78] H. Kopka and P. W. Daly. A Guide to LATEX. 3rd ed. Harlow, England:Addison-Wesley, 1999.

[79] C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[80] N. Kumar, B. R. Childers, and M. L. Soffa. Tdb: a source-level debugger for dynamically translated programs. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 123–132, 2005.

[81] 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*, SC '09, pages 58:1–58:12, New York, NY, USA, 2009. ACM.

[82] K. Lee, A. LaMarca, and C. Chambers. HydroJ: object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2003)*, pages 205–223. ACM, 2003.

[83] B. S. Lerner, H. Venter, and D. Grossman. Supporting dynamic, third-party code customizations in JavaScript using aspects. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA*, pages 361–376, 2010.

[84] O. Liechti, M. Sifer, and T. Ichikawa. Structured graph format: XML metadata for describing Web site structure. *Computer Networks and ISDN Systems*, 30(1-7):11–21, 1998.

[85] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329, New York, NY, USA, 1996. ACM.

[86] J. Liu and A. C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In *Proceedings of the 5th international symposium on Practical Aspects of Declarative Languages (PADL 2003)*, pages 110–127. Springer-Verlag, 2003.

[87] A. Marquez, S. Blackburn, G. Mercer, and J. N. Zigman. Implementing orthogonally persistent java. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, pages 247–261, London, UK, 2001. Springer-Verlag.

[88] V. Massol and T. Husted. *JUnit in Action*. Manning, 2004.

[89] C. Mastroianni, D. Talia, and P. Trunfio. Managing heterogeneous resources in data mining applications on grids using XML-based metadata. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 11, 2003.

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

[91] J. McGovern, R. Adatia, Y. Fain, J. Gordon, E. Henry, W. Hurst, A. Jain, M. Little, V. Nagarajan, H. Oak, et al. *Java 2 Enterprise Edition 1.4 (J2EE 1.4) Bible*. Wiley, 2011.

[92] Microsoft. Microsoft Open Database Connectivity. `http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx`.

[93] E. Miller. An introduction to the resource description framework. *Journal of Library Administration*, 34(3):245–255, 2001.

[94] T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for Java. *ACM Trans. Program. Lang. Syst.*, 31:7:1–7:54, 2009.

[95] 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*, PPPJ '02/IRE '02, pages 59–64, 2002.

[96] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove. X10 as a parallel language for scientific computation: Practice and experience. In *IPDPS: Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, 2011.

[97] Y. Minamide and A. Tozawa. XML Validation for Context-Free Grammars. In *Proceedings of the 4th ASIAN Symposium on Programming Languages and Systems (APLAS 2006)*, pages 357–373. Springer, 2006.

[98] K. Morik and M. Scholz. The miningmart approach to knowledge discovery in databases. In *In Ning Zhong and Jiming Liu, editors, Intelligent Technologies for Information Analysis*, pages 47–65. Springer, 2003.

[99] C. Noguera and L. Duchien. Annotation framework validation using domain models. *Lecture Notes in Computer Science*, 5095:48–62, 2008.

[100] N. Nystrom and V. Saraswat. An annotation and compiler plugin system for X10. Technical report, RC24198, IBM TJ Watson Research Center, 2007.

[101] 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. *LAMP-EPFL*, 2004.

[102] A. Orso, M. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *2nd Int. Workshop on Engineering Distributed Objects (EDO 2000)*. Springer.

[103] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, May 2005.

[104] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, October 2002.

[105] T. J. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th International Conference on World Wide Web, WWW*, pages 224–233, 2004.

[106] J. H. Perkins. Automatically generating refactorings to support API evolution. In *PASTE '05: Proceedings of the* 6th *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 111–114, New York, NY, USA, 2005. ACM.

[107] M. Philippsen and M. Zenger. JavaParty–transparent remote objects in Java. *Concurrency Practice and Experience*, 9(11):1225–1242, 1997.

[108] L. Popa, Y. Velegrakis, M. A. Hernández, R. J. Miller, and R. Fagin. Translating web data. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 598–609. VLDB Endowment, 2002.

[109] G. Pothier and É. Tanter. Extending omniscient debugging to support aspect-oriented programming. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 266–270, 2008.

[110] J. S. Poulin, J. M. Caruso, and D. R. Hancock. The business case for software reuse. *IBM Systems Journal*, 32:567–594, 1993.

[111] S. C. Previtali and T. R. Gross. Dynamic updating of software systems based on aspects. *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 83–92, September 2006.

[112] T. Ravichandran. Special issue on component-based software development. *SIGMIS Database*, 34:45–46, 2003.

[113] T. Ravichandran and M. A. Rothenberger. Software reuse strategies and component markets. *Commun. ACM*, 46:109–114, 2003.

[114] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3), 2007.

[115] C. Richardson. Untangling enterprise Java. *ACM Queue*, 4(5):36–44, 2006.

[116] S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2002.

[117] T. Ruotsalo and E. Hyvönen. An event-based approach for semantic metadata interoperability. In *ISWC'07/ASWC'07: Proceedings of the 6th international The semantic web and 2nd*

Asian conference on Asian semantic web conference, pages 409–422, Berlin, Heidelberg, 2007. Springer-Verlag.

[118] C. Russell. Java Data Objects 2.1, June 2007. `http://db.apache.org/jdo/specifications.html`.

[119] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specication Version 2.1. IBM Research, 2011.

[120] K. Schmid. A comprehensive product line scoping approach and its validation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 593–603. ACM, 2002.

[121] Shigeru Chiba. Java Programming Assistant. `http://www.csg.is.titech.ac.jp/~chiba/javassist`.

[122] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–38, 2004.

[123] Y. Smaragdakis and D. Batory. Application generators. *Encyclopedia of Electrical and Electronics Engineering*, 2000.

[124] Y. Smaragdakis, S. S. Huang, and D. Zook. Program generators and the tools to make them, 2004.

[125] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Bwl*, 147:195–197, 1981.

[126] M. Song. The structural enhancement rules language website. `http://research.cs.vt.edu/vtspaces/ser`.

[127] M. Song and E. Tilevich. The anti-goldilocks debugger: helping the average bear debug transparently transformed programs. In *OOPSLA Companion*, pages 811–812, 2009.

[128] M. Song and E. Tilevich. Enhancing source-level programming tools with an awareness of transparent program transformations. In *OOPSLA: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

[129] M. Song and E. Tilevich. Detecting metadata bugs on the fly. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1455–1456, 2012.

[130] M. Song and E. Tilevich. Metadata Invariants: Checking and inferring metadata coding conventions, 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pages 694–704, 2012.

[131] M. Song and E. Tilevich. Reusing non-functional concerns across languages, Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012. pages 227–238, 2012.

[132] Spring. Java web service. `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf`.

[133] E. Stark, M. Hamburg, and D. Boneh. Symmetric Cryptography in Javascript. In *Proceedings of Annual Computer Security Applications Conference, ACSAC*, pages 373–381, 2009.

[134] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656. Citeseer, 2005.

[135] H. Stuart, R. R. Hansen, J. L. Lawall, J. Andersen, Y. Padioleau, and G. Muller. Towards easing the diagnosis of bugs in os code. In *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*, pages 1–5, New York, NY, USA, 2007. ACM.

[136] R. Stuckert. JUnit reloaded, December 2006. `http://today.java.net/pub/a/today/2006/12/07/junit-reloaded.html`.

[137] Sun Developer Network. Java Adventure Builder Reference application. `http://java.sun.com/developer/releases/adventure/`.

[138] Sun Developer Network. Java Persistence API. `http://java.sun.com/javaee/technologies/persistence.jsp`.

[139] Sun Developer Network. Java Pet Store 2.0 reference application. `http://java.sun.com/developer/releases/petstore/petstore1_3_1_02.html`.

[140] Sun Microsystems. Java Platform Debugger Architecture. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[141] Sun Microsystems. JavaBeans Specification. `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html`.

[142] Sun Microsystems. The Java Database Connectivity. `http://java.sun.com/products/jdbc/overview.html`.

[143] Sun Microsystems Inc. Java 2 Platform, Enterprise Edition (J2EE), 2003.

[144] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 295–312, 2008.

[145] the AspectJ Team. The AspectJ 5 development kit developer's notebook. `http://eclipse.org/aspectj/doc/released/adk15notebook/index.html`.

[146] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 178–204. Springer-Verlag, LNCS 2374, 2002.

[147] E. Tilevich and Y. Smaragdakis. Binary refactoring: Improving code behind the scenes. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 264–273, May 2005.

[148] E. Tilevich and M. Song. Reusable enterprise metadata with pattern-based structural expressions. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 25–36, New York, NY, USA, 2010. ACM.

[149] R. Toledo, P. Leger, and É. Tanter. AspectScript: expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development, AOSD*, pages 13–24, 2010.

[150] T. Tourwé and T. Mens. Automated support for framework-based software evolution. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 148, Washington, DC, USA, 2003. IEEE Computer Society.

[151] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, 2006.

[152] E. Visser, Z. el Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *SIGPLAN Not.*, 34(1):13–26, 1999.

[153] C. Walls, N. Richards, and R. Oberg. *XDoclet in action*. Manning, 2004.

[154] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: aspect-oriented JavaScript programming framework for Web development. In *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS*, pages 31–36, 2009.

[155] C. A. White and M. Head-Gordon. Derivation and efficient implementation of the fast multipole method. *The Journal of Chemical Physics, Volume 101, Issue 8, October 15, 1994, pp.6593-6605*, 101:6593–6605, Oct. 1994.

[156] J. White, J. Hill, J. Gray, S. Tambe, A. Gokhale, and D. Schmidt. Improving Domain-Specific Language Reuse with Software Product Line Techniques. *Software, IEEE*, 26:47–53, 2009.

[157] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, 1997.

[158] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL*, pages 237–249, 2007.