Metadata Invariants: Checking and Inferring Metadata Coding Conventions

Myoungkyu Song and Eli Tilevich
Dept. of Computer Science
Virginia Tech, Blacksburg, VA 24061, USA
{*mksong, tilevich*}*@cs.vt.edu*

***Abstract*—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 paper 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. 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.**

***Keywords*-software maintenance; bug finding; refactoring; enhancement; frameworks; domain-specific languages; metadata; invariants.**

I. Introduction

A software application comprises functional and non-functional concerns. In a modern enterprise application, the programmer implements the functional concerns (i.e., business logic) procedurally by writing source code, and the non-functional 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 non-functional concerns (NFCs). Although the declarative programming model cleanly separates functional and non-functional 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 paper, 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 editor, 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 [11].

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 [21], [23].

This paper 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.

```java
1 public class ManagerModel {
2     private String orderId;
3     private String orderStatus;
4
5     // Other fields, getters, and setters go here.
6 }
```

```xml
1 <class name="ManagerModel" table="MANAGERMODEL">
2  <field name="orderId" column="ORDERID"/>
3  <field name="orderStatus" column="ORDERSTATUS"/>
4  <key column="ORDERID"/>
5 </class>
```

(1) XML deployment descriptor

```java
1 @Entity
2 @Table(name="MANAGERMODEL")
3 public class ManagerModel {
4     @Id
5     @Column(name="ORDERID", primaryKey="true")
6     private String orderId;
7
8     @Column(name="ORDERSTATUS")
9     private String orderStatus;
10
11    // Other fields, getters, and setters go here.
12 }
```

(2) Java 5 Annotation

Figure 1. Transparent Persistence Framework Example.

- 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.

The rest of this paper is structured as follows. Section II describes several program evolution scenarios that demonstrate how metadata invariants can help prevent bugs. Section III presents the Metadata Invariants Language. Section IV describes our inference algorithm. Section V presents the experimental results of evaluating our inference algorithm. Section VI outlines how we integrated metadata invariants with Eclipse. Section VII discusses related work. Section VIII presents future work directions and conclusions.

II. 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.

A. *Metadata-Related Bugs*

Consider a programmer enhancing a JUnit [22] 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 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 lower-left part of Figure 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 right part of Figure 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 paper 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 pat-

terns *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.

B. Finding Bugs Using Metadata Invariants

1) JUnit: Here we show how our approach can help find the bugs described above. Figure 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 III 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 annotated with `@TestSuite` 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.

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

Figure 2. Expressing JUnit 4 metadata invariants in MIL.

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 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 4 shows a skeletal example of a typical JUnit 4 test suite class. On line 9, 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 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.

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

Figure 3. MIL for JUnit 4 `void` methods.

```
1 ...
2 @TestSuite
3 class MyTestSuite {
4     @Before void setUpSuite() {...}
5     @After void tearDownSuite() {...}
6     @Test void testMethod1() {...}
7     @Test void testMethod2() {...}
8     ...
9     @Parameters
10    static java.util.Collection myData() {...}
11 }
```

Figure 4. A JUnit 4 test suite class.

```
1 ...
2   Method m in c
3   Where (m.returnType is Collection)
4    Assert(@Parameters m):
5      Msg("%s missing %s", m.name, @RunWith)
6 ...
```

Figure 5. Expressing JUnit 4 return type metadata invariant in MIL.

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.

2) Hibernate: Figure 6 shows the metadata invariant for the Hibernate [1] 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 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.

```
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",
8              @Column.name, f.name)
```

Figure 6. Expressing Hibernate annotations invariants in MIL.

Figure 7 shows a metadata invariant for Hibernate configured using XML. This invariant codifies the same naming convention as in Figure 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`.

```
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,
8       <class>.<field>.column)
```

Figure 7. Expressing Hibernate XML field column invariant in MIL.

The metadata invariant in Figure 8 verifies that all the fields of persistent classes are properly bound in the XML configuration file (e.g., the lower-left part of Figure 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.

```
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 8. Expressing Hibernate XML field name invariant in MIL.

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.

III. 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.

A. Language Summary

Figure 9 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.

B. Assertion Semantics

Next we describe how MIL expresses how metadata invariants are to be checked by evaluating various assertions. Figure 10 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

```
• 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.
```

Figure 9. MIL constructs and grammar.

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 pattern P_e

Figure 10. Syntax definitions.

$$\frac{Match(c,\ P_c)\ (P_c,\ \frac{M(a)}{t}) \in MIL}{t \in M_C(c)} \quad [\ Assert\ Class\ Metadata\ ]$$

$$\frac{Match(m,\ P_m)\ (P_m,\ \frac{M(a)}{t}) \in MIL}{t \in M_M(m)} \quad [\ Assert\ Method\ Metadata\ ]$$

$$\frac{Match(p,\ P_p)\ (P_p,\ \frac{M(a)}{t}) \in MIL}{t \in M_P(p)} \quad [\ Assert\ Parameter\ Metadata\ ]$$

$$\frac{Match(f,\ P_f)\ (P_f,\ \frac{M(a)}{t}) \in MIL}{t \in M_F(f)} \quad [\ Assert\ Field\ Metadata\ ]$$

Figure 11. Metadata invariants assertion rules.

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., $\star$) 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 [14].

Figure 11 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.

IV. 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.

A. 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 ($\star$) 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$\star$ in the MIL specification of their metadata invariant (e.g., **Where** (@Ann private $\star$ someName$\star$))[1]. Because we only use the $\star$ 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 [7]. 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 $\star$ character seems to be sufficient to express in a general form all the metadata invariants we have discovered.

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 : {α0, α1, .., αi, .., αn}
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
```

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

[1]The first $\star$ 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
4  is 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
```

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 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.

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
```

B. Implementation

Figure 12 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.

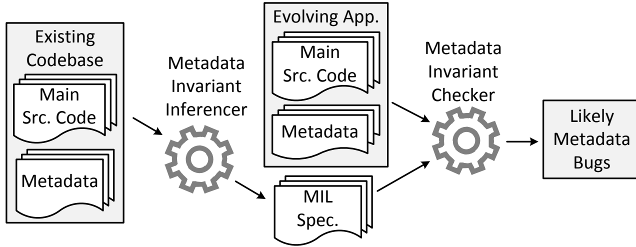

Figure 12. Integrating Metadata Invariants into Software Development.

In our implementation, we leverage common compiler backend techniques. We walk abstract syntax trees to infer and check metadata invariants as shown in Figure 13. To construct such 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.

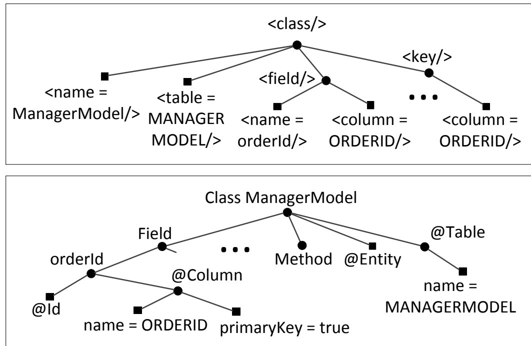

Figure 13. The generated tree structures for XML and Java 5 annotations (simplified version).

By walking the constructed abstract syntax tree, our implementation collects all the metadata related program constructs into a data repository as shown in Figure 14. 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.

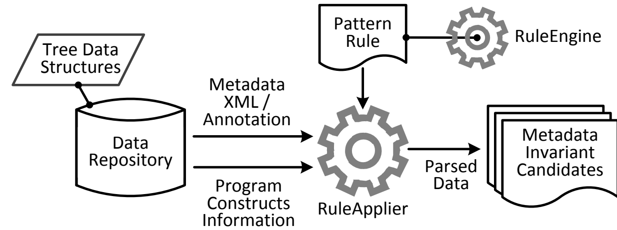

Figure 14. Discovering metadata invariant candidates.

Figure 15 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 applies these patterns on the ASTs to determine where the program violates the invariants and to report the violations to the programmer.

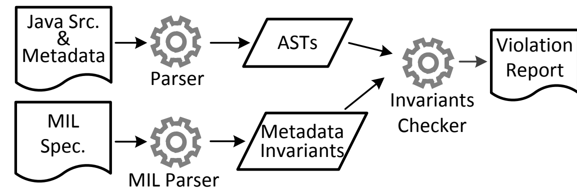

Figure 15. Checking metadata invariants.

V. Case Studies: 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[4], JEdit[5], Spring framework[6], JBoss Seam framework[7], IntelliJ

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

[4]Hibernate library – http://www.hibernate.org/
[5]JEdit text editor – http://www.jedit.org/
[6]Spring framework – http://www.springsource.org/
[7]JBoss Seam framework – http://www.seamframework.org

Table I
INFERRING METADATA INVARIANTS FROM THIRD-PARTY APPLICATIONS.

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** (@TestSuite * class *) **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))

IDEA[8], RunaWFE[9] and OpenVPN ALS[10]. For each subject application, we ran our metadata inference algorithm with a threshold of 96%.

Table I 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.

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 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.

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`

[8]IntelliJ IDEA – http://www.jetbrains.com/idea/
[9]Enterprise business process manager – http://wf.runa.ru/
[10]OpenVPN ALS server – http://openvpn-als.sourceforge.net/

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.

Table I shows how our approach 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.

VI. Integration with Eclipse IDE

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 our approach 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, our approach 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.

VII. Related Work

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

A. Validation for Metadata

Metadata invariants share the objective of validating the correctness of metadata with several prior efforts. Eichberg et al. [5] 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. [19] 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. [4] 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 [20] 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. [18] 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.

B. Pattern-Matching Techniques

In the domain of XML processing, techniques have been proposed [2], [3], [9], [10] to extract general XML programming patterns. However, only the patterns that occur in XML files are considered, not the ones that codify the relationship between metadata and the tagged program constructs.

Pattern-based reflective declaration [6], [12], [13] 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 [15]–[17]. 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.

C. Code Generation

Code generators have been employed to automatically synthesize metadata from higher level input. XDoclet, an extensible code generator [24], 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 [8] 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.

VIII. Future Work and Conclusions

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 [**?**].

In this paper, we presented metadata invariants, a novel approach to verifying metadata consistency and correctness. 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.

Acknowledgments

We would like to thank Wesley Tansey and the anonymous ICSE reviewers for their valuable feedback that helped improve the manuscript.

Availability

All the software described in the paper is available from: http://research.cs.vt.edu/vtspaces/metadata_invariants/.

References

[1] C. Bauer and G. King. *Hibernate in Action (In Action series)*. Manning Publications Co., 2005.

[2] 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.

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

[4] V. Cepa and M. Mezini. Declaring and enforcing dependencies between .NET custom attributes. In *Proceedings of the 3rd international conference on Generative Programming and Component Engineering (GPCE 2004)*, pages 283–297. ACM, 2004.

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

[6] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *Proceedings of the 5th international conference on Generative Programming and Component Engineering (GPCE 2006)*, pages 275–284. ACM, 2006.

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

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 26th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2005)*, pages 213–223. ACM, 2005.

[9] 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.

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

[11] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39:92–106, 2004.

[12] S. S. Huang and Y. Smaragdakis. Class morphing: Expressive and safe static reflection. In *Proceedings of the 29th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2008)*, pages 79–89. ACM, 2008.

[13] S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP 2007)*, pages 399–424. Springer-Verlag, 2007.

[14] 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 2001)*, pages 327–353. Springer-Verlag, 2001.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] C. Noguera and L. Duchien. Annotation framework validation using domain models. In *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA 2008)*, pages 48–62. Springer-Verlag, 2008.

[20] A. Orso, M. J. Harrold, and D. S. Rosenblum. Component Metadata for Software Engineering Tasks. In *Revised Papers from the 2nd international workshop on Engineering Distributed Objects (EDO 2001)*, pages 129–144. Springer-Verlag, 2001.

[21] M. Song and E. Tilevich. Reusing Non-Functional Concerns Across Languages. In *Proceedings of the 11th international conference on Aspect-Oriented Software Development (AOSD 2012)*. ACM, 2012.

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

[23] E. Tilevich and M. Song. Reusable enterprise metadata with pattern-based structural expressions. In *Proceedings of the 9th international conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 25–36. ACM, 2010.

[24] C. Walls, N. Richards, and R. Oberg. *XDoclet in Action (In Action series)*. Manning Publications Co., 2003.