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 program configuration files or 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.
JUnit 4 require that test methods be annotated with @Test with another test method. The programming conventions of
A. Metadata-Related Bugs
bugs from being introduced.
then show how our approach can prevent these and similar program evolution can introduce metadata-related bugs. We
Section VI outlines how we integrated metadata invariants experimental results of evaluating our inference algorithm.
Section IV describes our inference algorithm. Section V presents the describe how metadata invariants can help prevent bugs. Section
substrate how metadata invariants can help prevent bugs. Section III presents the Metadata Invariants Language. Section IV
describes how metadata invariants can help prevent bugs. 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-

(1) XML deployment descriptor
Figure 1. Transparent Persistence Framework Example.

(2) Java 5 Annotation
tells 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.

```
Invariant JUnitAnnotations<Package p>
  Class c in p
  Where (@TestSuite * class *)
  Method m in c
  Where (public void *)
  Assert (@Test m):
    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 straightforward to refine. Figure 3 shows a refined assertion on line 6 that includes all the possible annotations for public void test suite methods.

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

Figure 3. MIL for JUnit 4 void methods.

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 likely a 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.

```
   Invariant HibernateAnnotations<Package p>
   Class c in p
   Where (@Table public class *)
   Field f in c
   Where (@Column private * *)
   Assert (@Column.name eq f.name):
   Msg("%s will not be persisted", f.name)

Figure 6. Expressing Hibernate annotations invariants in MIL.
```

Next we describe how MIL expresses how metadata invariants are to be checked by evaluating various assertions. Before 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 mismatches orderStatus.

```
   Invariant HibernateXMLFieldName<Package p>
   Class c in p
   Where (<class>.name eq c.name)
   Field f in c
   Where (<class>.<field>.name eq f.name)
   Assert(<class>.<field>.column eq Uc(f.name)): 
   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
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 [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: \texttt{Where} (\texttt{m.returnType} is Collection). In addition, our algorithm uses the wild-card character (\texttt{\_}) to generalize the candidates based on their naming correspondences. For example, if all the \texttt{private} fields in a program tagged with the \texttt{@Ann} annotation have the names such as \texttt{someNameAbcdef, someNameGhij, someNameKlmm}, etc., the generalization (lines 9-13) will express the field names as \texttt{someName\_*} in the MIL specification of their metadata invariant (e.g., \texttt{Where (@Ann private + someName\_*)}). 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 [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 \_ character seems to be sufficient to express in a general form all the metadata invariants we have discovered.

\begin{algorithm}
\textbf{Algorithm 1: FindInvariantCandidates}
\begin{algorithmic}[1]
\State Input: Program \texttt{p} 
\State Output: Candidate \texttt{C} = \{c_0, c_1, ..., c_m\}
\State \forall t \in \texttt{Metadata}.
\State Let \texttt{pc} be a program construct in \texttt{p}.
\State \textbf{ForEach} \texttt{pc} \textbf{with} \texttt{t}
\If {\texttt{\_pc:[pc,t]}}
\State \texttt{A} = \{a_0, a_1, ..., a_i, ..., a_n\}
\State \texttt{Add pc to A}
\EndIf
\State A \leftarrow \texttt{ReplaceCommonSupertype}(A)
\State \texttt{Do}
\State l \leftarrow \texttt{LongestCommSub}(A, C)
\State A \leftarrow C
\State C \leftarrow \texttt{ReplaceWithWildCard}(pc, t, l)
\State Until \texttt{l} = 0
\EndDo
Return \texttt{C}
\end{algorithmic}
\end{algorithm}

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 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.

\begin{algorithm}
\textbf{Algorithm 2: VerifyInvariantCandidate}
\begin{algorithmic}[1]
\State Input: Candidate \texttt{c}, Program \texttt{p}, Threshold \texttt{\_}\texttt{e}
\State Output: Boolean \texttt{\_}\texttt{\_lambda}
\State \forall t \in \texttt{Metadata}.
\State Let \texttt{pc} be a program construct in \texttt{p}.
\State Let \texttt{alpha} and \texttt{beta} be the \#’s an invariant candidate is confirmed or disproven, respectively.
\State \textbf{ForEach} \texttt{pc} \textbf{with} \texttt{t}
\If {\texttt{\_pc:[pc,t]}}
\State \texttt{alpha} \leftarrow \texttt{alpha} + 1
\Else
\State \texttt{beta} \leftarrow \texttt{beta} + 1
\EndIf
\EndFor
\Return \texttt{\_\_lambda} = \texttt{beta/alpha} \leq \texttt{\_\_e}
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\textbf{Algorithm 3: CheckingMetadataInvariants}
\begin{algorithmic}[1]
\State Input: Program \texttt{p} MetadataInvariant \texttt{\_\_mu}
\State Output: Violations \texttt{V} = \{v_0, v_1, ..., v_n\}
\State \forall t \in \texttt{Metadata}; \forall a \in \texttt{Attribute} of \texttt{t}.
\State Let \texttt{pc} be a program construct in \texttt{p}.
\State \textbf{ForEach} \texttt{pc} \textbf{with} \texttt{t}
\If {\texttt{\_pc:[pc,t] | -\_\_mu}}
\State \texttt{Add pc to V}
\EndIf
\EndFor
\Return \texttt{V}
\end{algorithmic}
\end{algorithm}

B. Implementation

Figure 12 outlines how metadata invariants can be incorporated into a software development process. First, the metadata invariant inferrer 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.

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.

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\textsuperscript{4}, JEdit\textsuperscript{5}, Spring framework\textsuperscript{6}, JBoss Seam framework\textsuperscript{7}, IntelliJ

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

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

\textsuperscript{2}Eclipse Java development tools – http://www.eclipse.org/jdt.
\textsuperscript{3}Simple API for XML (SAX) – http://www.saxproject.org.
Table I
INFERRING METADATA INVARIANTS FROM THIRD-PARTY APPLICATIONS.

<table>
<thead>
<tr>
<th>Metadata Type</th>
<th>Application</th>
<th>Application Size (Files / LOC)</th>
<th>Inferred Metadata Invariant in MIL</th>
</tr>
</thead>
</table>
| Annotations   | Hibernate core 4.0.0 | 3,057 / 283,653 | Class c in p  
Where (@TestSuite * class *)  
Method m in c  
Where (@AFTER m)  
Assert (c,super has m) |
|               | JEdit 4.3.3 | 531 / 109,548 | 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 | Class c in p  
Method m in c  
Where (@Override m)  
Assert (c,super has m) |
|               | JBoss Seam 3.0.0 | 1,228 / 69,505 | Class c in p  
Field f in c  
Where (@XmlType.propOrder has f.name)  
AssertExists (@XmlType.propOrder has f.name) |
|               | IntelliJ 10.5.0 | 27,901 / 1,913,330 | Field f in c  
Where (@Attribute f)  
AssertExists (@Attribute.name) |
| XML           | RunaWFE 3.4.0 | 1,585 / 111,012 | Class c in p  
Where (@Attribute f)  
AssertExists (@Attribute.name) |
|               | OpenVPN ALS 0.9 | 1,954 / 165,000 | Class c in p  
Where (public class *)  
AssertExists (@Attribute.name) |

IDEA\textsuperscript{8}, RunaWFE\textsuperscript{9} and OpenVPN ALS\textsuperscript{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 @XmlElement 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.
attribute of the 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-extendible 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.

Minamida 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 [2].

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


