Using Annotations to Check Structural Properties of Classes

Michael Eichberg, Thorsten Schäfer, and Mira Mezini

Software Technology Group,
Department of Computer Science,
Darmstadt University of Technology, Germany
{eichberg, schaefer, mezini}@informatik.tu-darmstadt.de

Abstract. The specification of meta-information, by using attributes in .NET or annotations in Java, along with the source code is gaining widespread use. Meta-information is used for different purposes such as code generation or configuration of the environment in which a class is deployed. However, in most cases using an annotation also implies that constraints, beyond those defined by the language's semantics, have to be followed. E.g., a class must define a no-arguments constructor or the parameters of a method must have specific types. Currently, these constraints are not checked at all or only to a very limited extend. Hence, a violation can remain undetected and result in deployment-time or even subtle run-time errors. In this paper, we present a user-extensible framework that enables the definition of constraints to check the properties of annotated elements. Further, we demonstrate the application of the framework to check the constraints defined in the EJB 3.0 specification, and an evaluation of the approach based on checking the xPetstore-EJB3.0 project from within Eclipse to test the performance.

1 Introduction

The term *meta-information* refers to information about other information. In the context of programming languages it denotes information about program elements, which in turn represent information about an application domain. Meta-information on program elements is generally used by runtime environments and tools.

In Java, numerous examples of proprietary mechanisms to add meta-information to programs exist. Examples are tags like `@author` or `@version` used by the Javadoc tool to generate the class documentation. A similar approach is used by other tools such as XDoclet[1], Commons Attributes[2], JBoss AOP[3], or SGen[4]. Another example of extensive use of meta-information in Java are the various XML files in technologies such as Enterprise JavaBeans (EJBs)[5], Java Data Objects (JDO)[6], or Java Management Extensions (JMX)[7]. This information is used to configure the environment in which a class is to be deployed.

Currently, standard mechanisms are emerging to add meta-information to source code. In C#, [8] meta-information for source code artifacts like classes, methods, fields, etc. can be specified by means of *attributes* and in J2SE 5.0 by means of *annotations* [9]. The Java specification specifies six built-in annotations, how to declare annotation types, how to annotate declarations, and how to read those annotations later on. In addition to built-in annotations, there is also support to create and use user-defined annotations. Each annotation is considered a Java modifier and can be applied to annotate package, type, constructor, method, field, parameter, and local variable declarations. An annotation has a type and defines zero or more member-value pairs, each of which associates a value with a different member of the annotation type. E.g., in the following example the declaration of the class `CategoryX` is annotated with the annotation `@Entity`, whose member `access` is set to `AccessType.FIELD`, i.e., the container should access the entity's state using field access:

```
@Entity(access = AccessType.FIELD) public class CategoryX {...}
```

J2SE 5.0 annotations will have a fundamental effect on the way we program in Java. This is indicated by current development efforts on future versions of standard libraries. Major upcoming Java standards such as EJB 3.0[10], JDO 2.0[11], Java Web Services[12], or JDBC 4.0[13] will heavily rely on annotations. Further, a specification request exists to develop a set of annotations that apply across a variety of individual J2SE and J2EE technologies [14]. In the context of these specifications, annotations will be used for different purposes such as driving code generation, or supporting configuration. The rationale for the fast and widespread adoption of annotations is the expectation that their use will make the development process of components more lightweight and will flatten the learning curve of the supporting technologies.

However, a fact that is overseen by these efforts is that the use of annotations often imposes certain implementation restrictions on the decorated program constructs. Consider, e.g., the `java.lang.Override` annotation of Java 5, which can be used to annotate non-abstract methods to state that they that must be overridden in any subclass. Since `java.lang.Override` is a built-in annotation the implied implementation restriction is enforced by the Java compiler.

This is, however, not true for user-defined domain-specific annotations. An example for such annotations are those that will be part of the EJB 3.0 specification. In EJB 3.0, beans can be written as Java classes annotated with the specified EJB annotations. Based on these annotations, the container will generate the corresponding home and remote interfaces and extract the configuration information it needs. However, the effect of annotating a bean with, e.g., entity should go beyond driving the generation of its interfaces and providing configuration information to the container. It should also mean that implementation restrictions implied by the annotation, as explicitly stated in the specification, should be checked for, just like restrictions implied by built-in annotations are enforced by the compiler. An example of such a restriction on an entity bean is: *"An enterprise bean must not use thread synchronization primitives..."*.

From the discussion so far, it follows that automated annotation-based checking of implementation restrictions is needed. The *contribution of the work presented in this paper* is to provide support for this need. We present a user-extensible tool to bind checks of implementation restrictions to specific annotations. The tool is the first application which builds upon Magellan [15] - a generic platform for cross-artifact information retrieval during the software development process. Magellan enables to define queries over a uniform representation of all artifacts of a software project by mapping the artifacts of a project to XML representations and storing them in a database. Then XQuery, a functional query language for XML documents, can be used to query the database.

We extend this generic platform to check implementation restrictions based on Java annotations. The extension employs a time-efficient evaluation of checks by enabling a two-step querying process. In the first step, queries are run that select those program elements that are of common interest for queries evaluated in the second step. Certain implementation restrictions apply, e.g., only to entity beans. A query of the first step will select all entity beans. The result of this query determines the context for queries of the second step which encode the logic for checking different implementation restrictions. Hence, information that is needed by multiple queries is evaluated only once for all queries that need this information. Since the queries evaluated in the first step define a context for the evaluation of the subsequent queries they are called *context-defining queries*.

We also demonstrate the applicability of the proposed approach. As a proof of concept, we implemented queries to check the implementation restrictions defined by the EJB 3.0 draft[1]. These checks serve two purposes: (1) they demonstrate that the query capabilities used in our approach are sufficient for practical purposes, (2) they were used to evaluate the performance of our tool by running them against the xPetstore-EJB3.0 project. The results of this evaluation indicate that the tool can be used to check restrictions for annotated declarations on-the-fly for small to mid-sized projects (< 100-200 project classes), that is while the checks are performed in the background it is possible to continue editing in the foreground. Propositions about bigger projects cannot be done currently since there are no such projects publicly available that already use Java annotations. We will, however, provide some insights with this regard later in the paper.

This paper is structured as follows. The following section discusses the data model and the query language XQuery. Then queries to check implementation restrictions are discussed in Sec. 3. In Sec. 4 the architecture of the tool is presented and how queries are evaluated. In Sec. 5 we evaluate the performance and memory characteristics of our tool to show the feasibility of our approach. Related work is discussed in Sec. 6. Sec. 7 summarizes the paper and shortly discusses areas of future work.

[1] A prototype of the tool, including the checkers, is available as an Eclipse plug-in and can be downloaded from `http://www.st.informatik.tu-darmstadt.de/pages/projects/Magellan`.

2 Data Model and Query Language

2.1 Data Model

In this section, we discuss the data model that is the basis for the development of checkers. Since one goal of our approach is to provide a user-extensible tool, the selection of a comparable easy to comprehend data model is crucial. Due to the widespread knowledge of XML technologies we decided to build it upon an XML representation. A second reason for choosing XML is the free availability of industry-strength query languages. However, choosing XML as the underlying data format is not sufficient on its own. Additionally, we had to decide what kind of data should be represented. For the representation of Java code basically two choices exist. Either an XML representation of the abstract syntax tree (AST) of the source code can be used or a byte code based representation. At a first glance a representation based on the AST might look advantages because it is closer to what a standard Java programmer is used to. However, a bytecode representation has two advantageous. First, bytecode is less variform. E.g., in Java a field can be initialized directly, in an initializer, or in a constructor, but in bytecode all fields are initialized in a constructor. Hence, in bytecode the number of different cases how certain functionality can be expressed is smaller. This makes the development of checkers easier because it is not necessary to take multiple different possibilities into account. A second important point against choosing an AST-based representation is that checking an implementation restriction might require access to pre-built libraries that are not always available in source code (e.g., to determine inter-class relationships); so, some integration with a byte code representation would be needed anyway.

Our decision was to use an XML representation of the bytecode which is generated by BAT$_2$XML [16]. As a result the development of a checker might require some knowledge about Java bytecode and its XML representation in particular. Let's make an example to show how the XML database containing a representation of a Java class looks like. Assume we have the following class which declares a variable and a method, and uses annotations:

```
package xpetstore.domain.catalog.ejb;

@javax.ejb.Entity public class Category implements Serializable{
  private Long categoryId;

  @javax.ejb.Id public Long getCategoryId() {
    return categoryId;
  }
}
```

Listing 1.1. Category.java

The XML representation of this class, generated by BAT$_2$XML, is shown in listing 1.2 from line 6 to line 28.[2] The class itself is represented by the class element in line 6, while the method (line 16) is represented by a method element, and a field by a corresponding element (line 14). The attributes of these elements are self-explaining and define the properties of the declarations. The implementation of the method is shown in line 22 - 25; the field read access (`categoryId`) is represented by the get element (line 23).

```
 1 <db:all>
 2 <db:document type="source"
 3   documentID="file:/[PATH]/xpetstore/domain/catalog/ejb/Category.class"
 4   tag="de.tud.xirc.processor.input.ClassFileInputProcessor" >
 5
 6   <class
 7        name="xpetstore.domain.catalog.ejb.Category" visibility="public" ...>
 8     <annotations> <runtime_visible>
 9        <annotation type="javax.ejb.Entity"/>
10                        </runtime_visible> </annotations>
11     <inherits> <class name="java.lang.Object"/>
12                <interface name="java.io.Serializable"/> </inherits>
13
14     <field type="java.lang.Long" name="categoryId" visibility="private" .../>
15
16     <method name="getCategoryId" visibility="public" ...>
17       <annotations> <runtime_visible>
18          <annotation type="javax.ejb.Id"/>
19                          </runtime_visible> </annotations>
20       <signature> <returns type="java.lang.Long"/> </signature>
21       <code>
22         <load index="0" />
23         <get declaringClassName="xpetstore.domain.catalog.ejb.Category"
24              fieldName="categoryId" type="java.lang.Long" />
25         <return />
26       </code>
27     </method>
28   </class>
29 </db:document>
30 </db:all>
```

Listing 1.2. XML representation of the byte code of the `Category` class in the database

The `db:all` element (line 1) is the root element of the database and the `db:document` element (line 2 - 4) is used to structure all documents in the

[2] The compiler generated default constructor is omitted for brevity.

database. Its attributes define necessary information that are required for maintaining the database (line 3) and to enable further processing of query results (line 4).

2.2 Query Language

After choosing the data format we decided to use XQuery to implement the checkers. XQuery [17] is a query language especially well suited for XML data sources. While XQuery is a functional language comprised of several kinds of expressions that can be nested and composed with full generality, we will only elaborate on the features relevant to this paper. The most important among them is the notion of *path expressions*[3]. In a nutshell, a path expression selects nodes in a (XML-)tree.

For illustration, consider the previous XML document (Listing 1.2). We can parse this document by accessing the top-level document node (`db:all`) of the corresponding tree. Then the path expression `db:all/db:document/class/method/code/get` selects the `get` nodes, resulting in the node spanning line 23 to line 24 in Listing 1.2.

In general, a path expression consists of a series of *steps* separated by the slash character. The previous path expression has the steps, namely the *child* steps, `db:all`, `db:document`, `class`, `method`, `code` and `get`. The result of each path expression is a sequence of nodes. XQuery supports different directions in navigating through a tree, called *axes*. In the path expression above, we have seen the *child* axis. Other axes that are relevant for this paper are the *descendant axis* (denoted by "`//`"), the *parent axis* (denoted by "`..`"), the *ancestor axis* (denoted by "`ancestor::`") and the *attribute axis* (denoted by "`@`"). Using the descendants/ancestor axis rather than the child/parent axis means that one step may traverse multiple levels of the hierarchy. For example, the above query could be rewritten as: `//get`.

The attribute axis selects an attribute of the given node, whereas the parent axis selects the parent of a given node. For example, the path expression `//method/../@name` selects the `name` attribute of the declaring class of a method. Another important feature of XQuery is its notion of *predicates* – (boolean) expressions enclosed in square brackets to filter a sequence of values. For instance, the query `//method[@name="getCategoryId"]` selects all methods with the name `getCategoryId`. One can bind query results to variables, which in XQuery are marked with the `$` character, by means of a `let` expression, as illustrated below.

```
let $entityAnnotations := //annotation[@type="javax.ejb.Entity"]
return $entityAnnotations/ancestor :: class [@final = "true"]
```

XQuery also offers a number of operators to combine sequences of nodes, namely `union`, `intersect` and `except`, with the usual set-theoretic denotation,

[3] This subset of XQuery is a separate standard called XPath [18].

except that the result is again a sequence in document order, if required. The last relevant feature of XQuery is its notion of a function definition. For illustration, the function `directSupertypes` is shown below, which, being passed a set of class definitions, returns the classes that are directly inherited.

```
declare function xirc : directSupertypes ( $classes as element()*) as element()* {
  db: all /db:document/(class| interface )
                        [@name =$classes/inherits/( class | interface )/@name]
};
```

3 Checking Implementation Restrictions

In the following, we exemplary discuss the implementation of some checks on top of the discussed data model and query language to illustrate the possibilities offered by our approach, and to give an idea how to define new checks. Basically, a checker is just a query that selects elements which violate a restriction. Let us consider a simple check first. The EJB 3.0 draft specification[10] states in section 6.1 [Requirements on the Entity Bean Class] that:

> *The entity bean class must not be final. No methods of the entity bean class may be final.*

A possible checker is shown in the next listing. The first line selects all classes that have the `javax.ejb.Entity` annotation and stores the result in the variable `$ebs`. The variable `$xirc:project-files` is the set of all classes that are not defined in a library (`.jar` file). After that, line two determines for all entity beans (`$ebs`) the set of classes and methods that are declared final.

```
1 let $ebs := $xirc : project − files / class [./ annotations//@type ="javax.ejb. Entity"]
2 return $ebs[ @final = "true"] union $ebs/method[@final ="true"]
```

Certain annotations can only be used in combination [19]. E.g., annotating a method with `javax.jws.WebMethod` requires that the class is annotated with `javax.jws.WebService`[12]. To check this dependency the following query first selects all classes that declare a method with the `WebMethod` annotation (line 1) and then subtracts (line 2) all classes that are annotated with the `WebService` annotation (line 3). The set of classes that have `WebMethod`s but do not declare to be a `WebService` is returned.

```
1 $xirc : project − files / class [.// annotations//@type ="javax.jws.WebMethod"]
2 except
3 $xirc : project − files / class [./ annotations//@type ="javax.jws.WebService"]
```

The queries discussed so far could also be implemented using Java reflection, though the corresponding Java implementation would be harder to read and would require more effort: Explicit iteration over all classes and methods and checking each class' and method's modifiers. Fully checking the following restriction is no longer possible using Java Reflection because it requires information about a method's implementation, which is not exposed by Java Reflection. The EJB 2.1 specification (which is referenced by EJB 3.0) states in section 25.1.2 [Programming Restrictions]:

> *An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances.*

The following query checks that (a) no method is synchronized (line 2), that (b) `synchronize` is not used (line 3) - using Java's `synchronize` statement manifests in `monitorenter` and `monitorexist` instructions at Java bytecode level -, and that (c) none of the `wait` or `notify` methods is called (line 4 - 7).

```
1 let $c := $xirc:enterprise-beans()
2 return   $c/method[@synchronized="true"]
3          union $c/method/code//monitorenter
4          union $c/method/code//invoke[@declaringClassName="java.lang.Object"
5                   and (@methodName="wait" or
6                   @methodName="notify" or @methodName="notifyAll")]
```

The queries discussed so far are self-containing, i.e. the queries can be executed as is against the database. However, during the development of the EJB 3.0 checkers we realized that many queries have identical parts. E.g., the queries to check an entity bean's implementation restriction nearly always started with a path expression to determine all classes that are entity beans:

```
let $ebs := $xirc:project-files/class[./annotations//@type ="javax.ejb.Entity"]
```

Even more important, these parts required a significant amount of a query's evaluation time: In the case of a simple query up to 80-90%. To improve the performance of the query evaluation as well as to support a better modularization of the common part of queries we introduce context-defining queries. A context-defining query is a standard XQuery query where each node in its result set defines a context node for the subsequent evaluation of other queries. This node is passed to the query and can be accessed by using the "." operator. Multiple queries for checking implementation restrictions together with one context defining query are defined in a so-called Query Container.

For example, in Listing 1.3 lines 3-9 define a context defining query, which selects all classes that are enterprise beans. For each enterprise bean returned by the context-defining query the query defined in lines 14 - 16, which represents an implementation restriction, is evaluated. At the beginning of line 15 the context node, i.e. a class that is an enterprise bean, is accessed and used to select a `finalize()` method, if present. The listing also shows how to associate additional information (line 11 - 13) with a query.

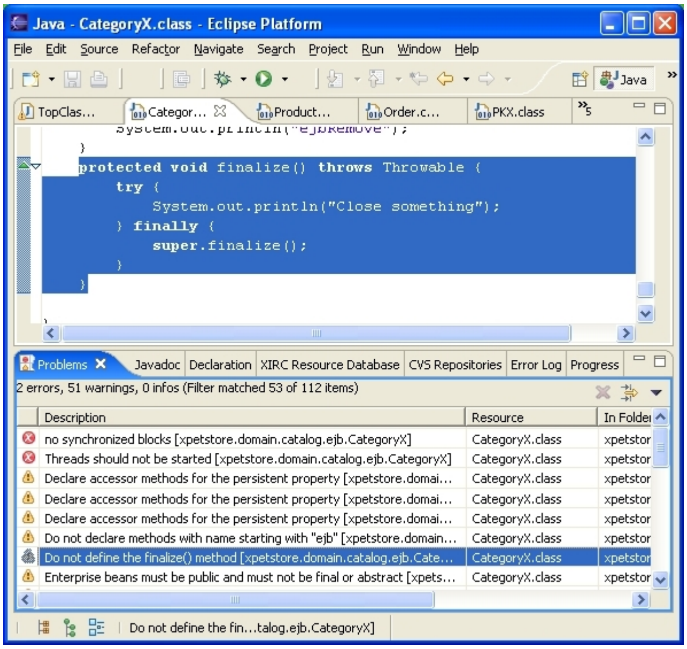
```
< implementation_restriction_container >
  < context_definition_type >query</context_definition_type>
  < context_definition >
    /db: all /db:document[@type = "source"]/class[
                              ./annotations//@type = "javax.ejb. Stateless "
                          or ./annotations//@type = "javax.ejb. Stateful "
                          or ./annotations//@type = "javax.ejb. Entity"
                          or ./annotations//@type = "javax.ejb.MessageDriven"
  </ context_definition >
  < implementation_restriction  id="FinalizeMethod" >
    <title>An enterprise bean must not define the  finalize () method.</title>
    <description>(see EJB 3.0  specification )</description>
    <severity>error</severity>
    <query>
      ./method[@name="finalize"' and empty(./signature/parameter)]
    </query>
  </ implementation_restriction > ...
</ implementation_restriction_container >
```

Listing 1.3. CommonEJB.XML; Query Container Definition

4 Architecture

As mentioned before, our tool is based upon Magellan [15], an open, cross-artifact information engineering platform integrated into the Eclipse IDE. Magellan provides the following services. Documents (in particular Java class files) are converted into corresponding XML-based representations and stored into a database. Changes to documents are tracked to keep the internal database up to date. In addition, a basic query facility is provided. When a client executes an XQuery query the corresponding XML nodes are returned as the result.

To check implementation restrictions our tool (XIRC) builds upon the Magellan platform and uses the provided services. For increased usability, XIRC is also developed as an Eclipse plug-in; however, the concept is also applicable to any other front-end, e.g., an integration with ANT. A user can enable the checking functionality on a project basis. If checking is enabled, the tool then creates special folders for managing the queries. Besides creating new query definitions in those folders and dropping existing query definitions in them it is also possible to include predefined checkers from a third party plug-in. This enables cost-effective reuse of checkers for common tasks, e.g., the checkers for EJB 3.0 are available as such a plug-in. Checking is triggered any time a resource, i.e. a Java class file or a checker, changes. Immediately after a change the Magellan plug-in synchronizes the database as discussed. Next, all queries found in the folder structure or provided by a plug-in are evaluated. The results of each query are passed to a special handler that is responsible for processing the resulting XML nodes. In this case, the handler maps the nodes back to the corresponding locations in the source code (e.g., to a class / method / field declaration or to a


Java - CategoryX.class - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java
TopClas... Categor... Product... Order.c... PKX.class
Problems Javadoc Declaration XIRC Resource Database CVS Repositories Error Log Progress

```
        System.out.println("ejbRemove");
    }
    protected void finalize() throws Throwable {
        try {
            System.out.println("Close something");
        } finally {
            super.finalize();
        }
    }
```

2 errors, 51 warnings, 0 infos (Filter matched 53 of 112 items)

	Description	Resource	In Folder
⊗	no synchronized blocks [xpetstore.domain.catalog.ejb.CategoryX]	CategoryX.class	xpetstor
⊗	Threads should not be started [xpetstore.domain.catalog.ejb.CategoryX]	CategoryX.class	xpetstor
⚠	Declare accessor methods for the persistent property [xpetstore.domai...	CategoryX.class	xpetstor
⚠	Declare accessor methods for the persistent property [xpetstore.domai...	CategoryX.class	xpetstor
⚠	Declare accessor methods for the persistent property [xpetstore.domai...	CategoryX.class	xpetstor
⚠	Do not declare methods with name starting with "ejb" [xpetstore.domain...	CategoryX.class	xpetstor
⚠	Do not define the finalize() method [xpetstore.domain.catalog.ejb.Cate...	CategoryX.class	xpetstor
⚠	Enterprise beans must be public and must not be final or abstract [xpets...	CategoryX.class	xpetstor

Do not define the fin...talog.ejb.CategoryX]

Fig. 1. Violations of constraints implied by annotations in Eclipse

line number in the source code). Additional information defined along with the query, such as the severity of the restriction or a problem description, are used to inform the developer about the broken restriction (see Figure 1).

Figure 1 shows an example of the results from multiple queries. In the lower half of it the standard problem view of Eclipse is shown with multiple violated restrictions for class `CategoryX`. The developer can see the severity, a description and the location where the violation occurs. ¿From the second last entry in the problems view it can be seen that it is possible to navigate to the corresponding location in the source code by selecting an entry.

5 Evaluation

Before we will discuss the performance, we first discuss the effort necessary when developing new checkers. We made the experience that the biggest effort when writing queries is to learn to use XQuery. The effort was not to understand the XML representation generated by BAT$_2$XML. This is probably due to the fact that most checkers do not require sophisticated control-flow or data-flow analysis and that it is sufficient to simply take a look at the XML representation of a class to write the query. A detailed knowledge of the execution semantics of bytecode instructions is not necessary. Hence, we expect that developers familiar with XQuery and Java can immediately start writing queries to check structural

Table 1. Evaluation times of queries

Short Description	Seconds
CommonEJB.xml	$\sum$ 0.643225
context defining query	*0.023961*
an EJB must not start threads	0.017519
signature of call back method is invalid	0.069257
the chosen transaction attribute cannot be used	0.011743
an EJB must have a no-arg constructor	0.010397
a business method must not start with “ejb”	0.012741
an instance that starts a transaction must complete the transaction before it starts a new transaction	0.385770
`(get|set)RollbackOnly` should be called only in bean methods that execute in the context of a transaction	0.044467
`UserTransaction` is unavailable to EJBs with CMT demarcation	0.011552
a `TransactionAttribute` can only be specified with CMT demarcation	0.012814
EJBs should not handle concurrent access on their own	0.013553
SessionEJB.xml	$\sum$ 0.831755
context defining query	*0.204696*
for update / delete operations a transaction context is required	0.047968
argument and return types must be legal types for RMI/IIOP	0.476183
argument and return types must be legal types for JAX-RPC	0.027315
multiple business interfaces should be annotated as `Local` or `Remote`	0.046658
this `SessionContext`'s method cannot be called	0.024672
EntityEJB.xml	$\sum$ 1.928637
context defining query	*0.147486*
persistent field has invalid type	0.463888
persistent properties with `@Basic` may not be an entity association	0.016634
invalid dependent class	0.159400
a protected field is to accessed by the defining class only	0.032637
an entity bean that is a subclass of another entity bean must have the same primary key	0.155760
entity beans must have getter/setter-methods for persistent fieds	0.099020
collection-valued persistent properties must have type `java.util.Collection` or `java.util.Set`.	0.737543
invalid type for primary key	0.080830
MessageDrivenEJB.xml	$\sum$ 0.015559
context defining query	*0.011637*

properties and that those checks can be written in a reasonable amount of time, that is implementing and testing a query requires less than an hour.

To assess the potential, performance, and memory consumption of our approach we have developed a full set of queries to check the constraints defined in the EJB 3.0 draft specification[10]. The queries were evaluated against a demo release of the xPetstore project[20][4] project that was updated by Bill Burke and Gavin King for EJB 3.0.

The following measurements were taken on an Intel Celeron 2.40 GHz system with 504 MB RAM running Windows XP and using J2SE 5.0, Saxon 8.1 and Eclipse 3.1M2 as the underlying platform. The XML database had 2833 class entries, which represented all public classes and interfaces of all Java APIs[5] delivered with Java 5, except for classes in the `javax.swing.*` and `java.awt.*` packages. Additionally, all necessary JARs to compile the xPetstore project were included. The evaluation for the original xPetstore project which run without any error being signaled required 1.97 seconds. To make the evaluation more realistic, we injected some more or less severe problems into the project code. The evaluation of all 48 queries against the messed project code generated correctly 53 messages and was executed in 3.56 seconds. In both cases, the time required by Eclipse to recompile the source file and to update the Magellan database should be added, which amounts to another 1-2 seconds. To keep the Magellan database in memory approximately 40 MB are required.

Detailed execution times are shown in Table 1; the table lists the times required to evaluate the query containers (printed bold) as a whole, as well as the times required for the evaluation of each context defining query, and the times for the queries to check the constraints along with a short description of the checked constraint. Queries that took less than 10 milliseconds to evaluate are omitted for brevity. The descriptions were shortened; the messages shown to developers are more detailed.

The result of this preliminary analysis shows that the overhead (less than five seconds and running in a non-blocking background process), generated by checking all implementation restrictions, is acceptable for a day-to-day usage. Further, the evaluation shows that the implementation restrictions defined by EJB 3.0 can be checked by using a declarative, though functionally complete, query language; it is not necessary to write the checks as imperative meta-programs in a "standard programming language" such as Java.

6 Related Work

The purpose of FindBugs[21] is to find bugs or potential bugs in existing projects based on control and data flow analysis. In contrast to our tool, FindBugs does not enable to write declarative queries. Instead, to detect a bug a visitor[22] has to be written that visits the in-memory representation of a class' bytecode and reports errors and warnings. JLint[23] and JiveLint[24] are further tools to detect bugs, which are similar in scope and functionality to FindBugs. However, while these tools are concerned with identifying general bugs that are independent of the usage of specific frameworks our approach is targeted at identifying specific implementation restrictions that need to be checked if and only if a specific framework is used.

[4] xPetstore-EJB3.0: http://cvs.sourceforge.net/viewcvs.py/jboss/xpetstore-ejb3.0/

[5] Classes starting with `com.*` are irrelevant for the checks and were not included.

IRC [25] is similar to FindBugs in the respect that a *checker* also analyses the in memory representation of a class' bytecode. But in contrast to FindBugs a sophisticated framework exists to programmatically construct queries to check the code. So, while evaluation speed is explicitly targeted by IRC writing a query still involves writing Java code and requires detailed knowledge of the internal representation of the byte code. Based on a comparison of the development of checkers using IRC and our new tool XIRC our experience is that writing, maintaining and evolving declarative queries on top of an XML representation is easier and can be done in less time. The development of checkers (for EJB 2.1) for IRC needed approximately double the time than the development for XIRC; though, the preconditions were comparable: The students who developed the checkers had no knowledge about the framework or the byte code representation in case of IRC and no knowledge about XQuery or the XML representation of byte code in case of this work.

AspectJ[26] can also be used for constraint checking[27]. However, AspectJ was not primarily designed to do it and, as we have argued in [25], the possibilities offered by static pointcuts to detect violations of constraints are too limited to be useful in general.

PMD[28] is similar to our tool in the respect that it also supports to write declarative queries by using XPath, which is an important part of the XQuery language. However, PMD operates on the abstract syntax tree of a program and its primary goal is to check the style of a program and not the semantics. In particular, the used abstract syntax tree does not contain resolved type information, e.g., the types of the formal parameters of a method are not available from looking at a method call node in the AST. This makes writing queries that take type information into relation or that need to span multiple classes tedious and error-prone. Checkstyle[29] is similar to PMD and suffers from the same problem.

The idea of Splint[30] is to annotate the source code (ANSI C) to make design decisions or implementation restrictions explicit. E.g., to annotate a parameter with `@notnull` to indicate that the parameter should never be null. Splint will then perform a static analysis of the code using the annotations and report violations. Splint is designed as a compiler; extensibility by users was not a goal. However, it would be an interesting exercise to develop a set of similar Java annotations and checks that can be used by developers to make implementation restrictions explicit in their code and which are checked.
ESC/Java2[31] also uses annotations of the Java source code to enable an extended static analysis. Since ESC/Java is based on theorem proving the evaluation times are very high [32]; on-the-fly evaluation is out of scope.

7 Summary and Future Work

With the standardization of annotations in J2SE 5.0, a common metadata facility is now available for the programming language Java. Forthcoming standards in the Java landscape such as EJB 3.0, JDBC 4.0 and Web Services Metadata show the widespread adoption of annotations. As argued previously, the usage of meta information in program code often implies that specific implementation restrictions have to be obeyed by the annotated declarations to guarantee that the program will work properly. Though, implementation restrictions are not new we argue that annotations represent perfect join points in the source code where to start checking restrictions.

We have shown that our tool can check structural properties of classes by using annotations, and that the checks themselves can be defined using declarative queries. For evaluation we applied our framework to the EJB 3.0 specification and, as our evaluation suggests, the performance is already good enough to use it for small to mid size projects. The tool is user-extensible and fully integrated into the Eclipse IDE enabling checks during the development process.

To the best of the authors knowledge, we presented a first fully-integrated tool which is capable of on-the-fly checking of properties based on Java's new annotation facility.

Currently, all queries are always evaluated against the entire database, which is reasonable fast for small to mid sized projects. But for large projects with hundreds of classes the achieved performance may be too slow; even though the evaluation is executed in the background, evaluation times beyond 10 to 15 seconds are not acceptable. The problem is that a long-running build process may prevent other (background-)processes from execution and may finally require the developer to stop the work until the processes have completed. To achieve faster evaluation times we are going to investigate queries that are evaluated per changed document, i.e., a changed document is set as the context node for the query evaluation. However, in this case it is necessary to keep track of all documents visited by a query in order to know when to reevaluate it. The question is, if the necessary effort for tracking and managing these information finally pays off.

Acknowledgments

The authors would like to thank Cuma Ali Gencdal, who implemented parts of the support for annotations in BAT_2XML and most of the queries for EJB 3.0.

References

1. Team, X.: XDoclet: Attribute-Oriented Programming. (http://xdoclet.sourceforge.net/)
2. Foundation, A.S.: Commons attributes. jakarta.apache.org/commons/attributes/ (2004)
3. Inc., J.: JBoss AOP 1.0 beta3. http://www.jboss.org (2004)
4. Beust, C.: SGen. http://www.beust.com/sgen/ (2004)
5. DeMichiel, L.G.: Enterprise JavaBeans Specification, Version 2.1. SUN Microsystems (2003)
6. Russell, C.: Java Data Objects, Version 1.0. SUN Microsystems (2002)

7. Sun Microsystems: Java management extensions. White paper, Palo Alto, California, USA (1999)
8. Archer, T.: Inside C#. Microsoft Press (2001)
9. Bloch, J.: A metadata facility for the java programming language. Java Specification Request 175, SUN Microsystems (2002)
10. DeMichiel, L.G.: Enterprise javabeans specification, version 3.0. Java Specification Request 220 (2003)
11. Russell, C.: Java data objects 2.0 - an extension to the jdo specification. Java Specification Request 243 (2004)
12. Zotter, B.: Web services metadata for the java platform. Java Specification Request 181 (2004)
13. Bruce, J.: Jdbc 4.0 api specification. Java Specification Request 221 (2004)
14. Mordani, R.: Common annotations for the java platform. Java Specification Request 250 (2004)
15. Eichberg, M., Mezini, M., Ostermann, K., Schäfer, T.: A kernel for cross-artifact information engineering in software development environments. In: Proceedings of 11th IEEE Working Conference on Reverse Engineering (WCRE), IEEE Computer Society (2004) to appear.
16. Eichberg, M.: Battoxml. http://www.st.informatik.tu-darmstadt.de/BAT (2004)
17. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: Xquery 1.0: an xml query language. Working Draft 23 Juli 2004, (W3C)
18. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. (http://www.w3.org/TR/1999/REC-xpath-19991116)
19. Cepa, V., Mezini, M.: Declaring and enforcing dependencies between .net custom attributes. In: Proceedings of the Third International Conference on Generative Programming and Component Engineering. (2004)
20. Tchepannou, H., McSweeney, B., Cooley, J.: xPetstore. http://xpetstore.sourceforge.net (2003)
21. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Notices **December** (2004)
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Professional Computing Series. Addison-Wesley (1995)
23. Artho, C.: Finding faults in multi-threaded programs. http://artho.com/jlint/ (2001)
24. Sureshot: JiveLint v1.22. (http://www.sureshotsoftware.com/javalint/)
25. Eichberg, M., Mezini, M., Schäfer, T., Beringer, C., Hamel, K.M.: Enforcing system-wide properties. In: Proceedings of the 15th australian software engineering conference (ASWEC), IEEE Computer Society (2004)
26. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: Proceedings of the 15th european conference on object-oriented programming (ECOOP). Volume 2072 of Lecture Notes in Computer Science., Budapest,Hungary, Springer (2001) 327–355
27. Shomrat, M., Yehudai, A.: Obvious or not? regulating architectural decisions using aspect-oriented programming. In Kiczales, G., ed.: Proceedings of 1st international conference on aspect-oriented software development (AOSD), Enschede, The Netherlands", ACM Press (2002) 3–9
28. PMD. (http://pmd.sourceforge.net)
29. Kühne, L., Studman, M., Burn, O., Sukhodolsky, O., Giles, R.: Checkstyle. http://checkstyle.sourceforge.net/ (2004)

30. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software **January / February** (2002)
31. Cok, D., Kiniry, J.: Esc/java2. http://www.cs.kun.nl/sos/research/escjava/ (2004)
32. Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for java. In: 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04). (2004) to appear.