Instrumenting Annotated Programs

Marina Biberstein
IBM Haifa Labs
Haifa University Campus
Haifa 31905, Israel
biberstein@il.ibm.com

Vugranam C. Sreedhar
IBM T.J. Watson Research Lab
19 Skyline Drive
Hawthorne, NY 10532, USA
sreedhar@watson.ibm.com

Bilha Mendelson
IBM Haifa Labs
Haifa University Campus
Haifa 31905, Israel
bilha@il.ibm.com

Daniel Citron
IBM Haifa Labs
Haifa University Campus
Haifa 31905, Israel
citron@il.ibm.com

Alberto Giammaria
IBM Austin
11400 Burnet Road
Austin, TX 78758, USA
agiammar@us.ibm.com

ABSTRACT

Instrumentation is commonly used to track application behavior: to collect program profiles; to monitor component health and performance; to aid in component testing; and more. Program annotation enables developers and tools to pass extra information to later stages of software development and execution. For example, the .NET runtime relies on annotations for a significant chunk of the services it provides. Both mechanisms are evolving into important parts of software development in the context of modern platforms such as Java and .NET.

Instrumentation tools are generally not aware of the semantics of information passed via the annotation mechanism. This is especially true for post-compiler, e.g., runtime, instrumentation. The problem is that instrumentation may affect the correctness of annotations, rendering them invalid or misleading, and producing unforeseen side-effects during program execution. This problem has not been addressed so far.

In this paper, we show the subtle interaction that takes place between annotations and instrumentation using several real-life examples. Many annotations are intended to provide information for the runtime; the virtual environment is a prominent annotation consumer, and must be aware of this conflict. It may also be required to provide runtime support to other annotation consumers. We propose an annotation taxonomy and show how instrumentation affects various annotations that were used in research and in industrial applications. We show how the annotations can expose enough information about themselves to prevent the instrumentation from accidentally corrupting the annotations. We demonstrate this approach on our annotations benchmark.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'05, June 11-12, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-047-7/05/0006...$5.00.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Standardization, Languages

Keywords

Program instrumentation, annotation, custom attributes, meta-annotation

1. INTRODUCTION

System management tools are becoming critical in the context of on-demand operating environments. Such tools widely rely on two enabling technologies: instrumentation and program annotation. Instrumentation is used in collecting profiling information that is needed for monitoring the performance, provisioning, or other execution characteristics. Program annotations are information passed by application developers or tools to the virtual runtime so as to specialize or tailor the execution characteristics of the application. For instance, a compiler can pass optimization information so as to improve application performance. Unfortunately, instrumentation tools are not aware of the semantics of information passed via the annotation mechanism, and if one is not careful, the interaction between instrumentation and program annotation can lead to unexpected behavior of the application. In this paper, we first show the subtle interaction that can take place between program annotation and instrumentation. A typical scenario during application development and execution is as follows: First, the application is annotated as part of the application development. Second, the program is instrumented to collect profiling information. Finally, either the run-time or a tool interprets the program annotation (of the instrumented program). We present a solution to address the interaction problem. Our approach is based on first studying the characteristics of existing annotations. Based on the study, we present a taxonomy of meta-annotation that can help alleviate the interaction problem between program annotation

and instrumentation. We think it is very important to address these issues while the usage of annotations is still in its infancy and there is time to apply the necessary changes.

Modern programming languages, as well as traditional ones, support program annotation. In Java and CLR, annotations constitute a powerful mechanism that enables passing information between programmers, tools, and the runtime, from the source code level up to the execution time. Program annotations enrich the program semantics and facilitate optimizations. They describe method usage (test markers, web method markers), convey optimization hints (static register allocation schemes, redundant runtime checks markup), or aid in code development and maintenance (author name tags, bug tracking). In Java, annotations were only standardized recently in release 1.5, while CLR incorporated them in the initial standard. The .NET experience shows that two-thirds (66%, or 143 out of 215) of the non-standard annotations implemented by this platform target the runtime, providing hints to serialization and remoting services, interoperation with native code, and more. There is also a growing body of annotations consumed at runtime by third-party tools such as test drivers [24].

Instrumentations are commonly used to track application behavior: to collect program profiles; to monitor component health and performance; to aid in component testing; and more. As opposed to general-purpose program transformations, instrumentation only aims to gather additional information about the system rather than modify the original program's structure and behavior. Bytecode instrumentation uses structural and semantic information provided by language and platform specifications both to identify instrumentation points and to avoid affecting the original program structure and behavior. Both Java and .NET provide (non-standard) profiling APIs for run-time program transformation and instrumentation. *Java Instrumentation Services* [11] are a recent addition that standardizes the means by which a Java "agent" can query and modify the JVM in which it is running, and in particular, instrument at runtime classes running in that JVM. The Instrumentation Services were standardized in the same 1.5 release of Java as the annotations — ironically, since, as we show in the following example, in the presence of program annotations with unknown functionality and semantics, an instrumentation could affect correctness of both the annotation and the instrumented program.

EXAMPLE 1. *Array boundary checks.* In Java, every array access must be accompanied by a reference nullity check and array boundary checks, to ensure that correct exceptions are thrown by the runtime if necessary. Qian et al. [30] suggested an analysis that identifies instructions where such checks are superfluous due to previous array accesses within the method. The results of this analysis are conveyed through annotations attached to the analyzed method. For example, `@array_null_check(59)` means that checks at bytecode offset 59 can be omitted.

Consider now a profiling tool that instruments the annotated class file to get basic block usage data. This is done by inserting some extra instructions in the begiining an end of each basic block, and, consequently, has the side effect of moving the original instructions within the code array. Now there may not be any instruction that starts at the offset 59 specified by the annotation above. Even worse, offset 59 may point to a valid instruction, but a different one. In this case, JIT or other annotation users have no indication that the annotation is invalid, and by using it, they change the program semantics and introduce security hazards.[1] ■

```
1 public class WebRead{
2   @WebMethod
3   @array_null_check(59)
4   public String ReadStory(String filename)
5   {
6    // Retrieve the story
7   }
8 }
```

Figure 1: A sample web method

In the example above, the instrumentation ignores the presence of the annotation, which leads to a semantics change. Is the solution for the instrumentation to remove the annotations it encounters? We discuss this in the following example.

EXAMPLE 2. *Web services.* Web services are gaining popularity as a way to integrate heterogeneous distributed applications or services. A client and a web service provider communicate using HTML- or SOAP-formatted messages. Web services are published using the Web Service Description Language (WSDL). In both J2EE and .NET, methods that implement web services need special support from tools and runtime. In .NET, web services are marked as such by the programmer using annotations (known in .NET as custom attributes): attaching the `@WebMethod` annotation to a public method converts it into a web method. Figure 1 illustrates the use of both `@WebMethod` and `@array_null_check` on the same method.[2]

Suppose that the code in Figure 1 is instrumented to create a method-level profile of the application. We have already seen that modifying the method code renders `@array_null_check` unusable. However, instead of placing the new code within the profiled method as in Example 1, the method-level instrumentation could move the profiled method's body into a new method as in Figure 2. `ReadStoryI()` is now an exact copy of what `ReadStory()` was before the instrumentation, so the instrumentation can move `@array_null_check` from `ReadStory()` to `ReadStoryI()` to avoid the correctness problems found in Example 1. Handling both annotations identically, the instrumentation moves `@WebMethod` as well.

Does this work? No: now `ReadStory()` is not a web method, since it is no longer annotated as such. `ReadStoryI()`, on the other hand, *is* a web method, but it is not published in the corresponding WSDL file. In order to fix this, the instrumentation would need to move the `@WebMethod` annotation back to `ReadStory()`, while leaving the `@array_null_check` annotation attached to the new `ReadStoryI()` method.

We see that the two annotations have to be handled differently. Deleting or moving `@WebMethod` affects the usability of the resulting code; however, it would work if left attached to the instrumented method. In contrast, deleting

[1]Had the instrumentation known the semantics of the offset field, it could have fixed the offset in the annotation, but recall that the semantics of annotation fields is generally unknown.

[2]To avoid confusion, we use the Java style (`@Annotation`) rather than C# style (`[Annotation]`) throughout this paper.

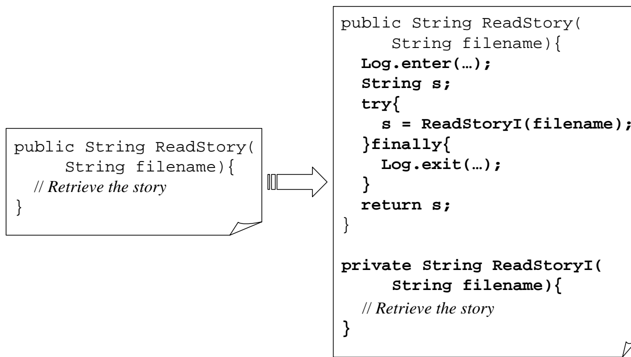

Figure 2: External instrumentation

`@array_null_check`, or, better, moving it with the method body works fine. But if the instrumentation just leaves it attached to the modified method, the combination of instrumentation and annotation can lead to severe errors and changes in program semantics. ■

Currently, there is no way for instrumentation to safely treat non-standard annotations, i.e., annotations that are not part of the language or platform standard. In particular, it is impossible for an instrumentation to reason about the semantics of non-standard annotations it encounters, since the only information available is on types of annotation fields, not their meaning. Given that annotation programming is becoming popular on modern programming platforms like Java and .NET, it is important to address the problem of interaction between annotations and program transformations, and especially instrumentations.[3]

To summarize, the main contributions in this paper are as follows:

- We expose problems that stem from applying instrumentation on annotated code, and illustrate them using real world examples.
- We introduce the notion of *annotation modes* to systematically classify the annotations. This classification, grouping together annotations whose behavior under instrumentation is similar, is used to analyze a representative set of annotations drawn both from the literature and from existing programs.
- We present a light-weight behavioral classification of annotations and show how annotators can use it for communication with the instrumentation.
- We present practical results from the application of our classification to the annotations in our benchmark set and show how an instrumentation uses this information to achieve safe code modification.

The rest of the paper is organized as follows: In Section 2, we present some background on annotations and instrumentations. Specifically, we survey popular instrumentation techniques, and discussing different varieties of annotations that have been explored in the past. In Section 3, we present a simple taxonomy for these annotations. In Section 4, we suggest how the annotation writer can provide the instrumentation with a description of relevant properties. We review the related work in Section 5, and wrap up with conclusions.

[3]Instrumentations, which gather monitoring information, are especially important because they are more often than, say, code patches, applied by parties that do not participate in the development of the affected code. Consequently, these parties do not possess any special knowledge about the semantics of the annotation used.

2. INSTRUMENTATION AND ANNOTATION

2.1 Instrumentation

As opposed to general-purpose program transformations, instrumentations, or *spectative* program transformations [33], strive to maintain the program structure and functionality, allowing only minor side effects such as increases in execution time or changes to the log file. Such instrumentations do not remove program elements (e.g., classes, fields, and methods); variables defined by the original program may be read but not written. Instrumentation may add its own variables, even to existing program elements (e.g., new fields or local variables), and those variables may be read or written by it. Instrumentation may also insert new code into original program methods, and invoke other methods from this code, provided that original variables are not modified as a result of these invocations; ensuring this is the responsibility of the instrumentation. Finally, instrumentation may *outline* code, i.e., move all or part of the method code into a new method and replace it in the original method with the invocation of the new one. The external instrumentation in Figure 2 is an example of out-lining.

There is a wealth of applications for program instrumentation. We have already mentioned fine-grained profiling; while the standard profiling APIs allow only method-level information, bytecode instrumentation has to be used, for example, to measure execution frequency of individual basic blocks. Another application area is that of system management tools that track characteristics of other applications: health, performance, workload, serviceability, license usage, etc. These tools require the managed application to be instrumented with system management APIs, e.g., ARM [26]. Instrumentation is also used by testing and verification tools. For example, JSpy [12] uses instrumentation to log certain kinds of operations; the log is later used for the verification of various program properties. ConTest [9] strives to diversify in the program's thread interleaving schedules during testing by inserting various semantics-neutral instructions such as `sleep()` and `yield()` into the method bodies. This is actually the purest kind of instrumentation, since there are no side effects and any change in behavior caused by the instrumentation cannot be distinguished from a change in behavior due to a different runtime environment.

2.2 Annotations

Pragmas were an early incarnation of metadata annotations, passing information from the programmer to the build tools. Other tools used source-file annotations (appearing either as new keywords or as specially-formatted comments) to receive hints from the programmer, for example, variable mutation and aliasing information [10, 1, 23]. Annotations guide compilers in concurrent code generation [21, 3] and declare method pre- and post-conditions [14, 16].

Java [20] permitted the use of annotations (under the name of *attributes*) at the class file level. In the early days

```
1   /** Describes the request-for-enhancement (RFE)
2       that led to the presence of API element*/
3   /* Attached to classes, fields, and methods*/
4   @Target(CLASS, METHOD, FIELD)
5   /* Appears in the class file, not at runtime*/
6   @Retention(CLASS)
7   public @interface RFE{int id(); String date();}
8
9   /* Annotation normal usage */
10  @RFE(id = 2868724, date = "4/1/2004")
11  public static void travelThroughTime(
12                  Date destination) { ... }
```

Figure 3: An example of Java source annotation

of Java, an attribute was a byte array of known length and unknown semantics attached to a class file element, e.g., the class itself, fields, methods, or code arrays. A virtual machine is required to silently ignore all attributes that it does not recognize, and the attributes are prohibited from affecting the semantics of Java types. However, if attributes expected by a tool are absent from the class file, or are no longer valid because of instrumentation, the tool may function incorrectly. Annotations on bytecode were applied in research, for example, to pass information from a static compiler to JIT [18, 30, 2, 13], or to store program analysis results [28].

One of the reasons that attributes never gained popularity in the industry was the absence of a convenient way for a Java programmer to create these attributes. Java 1.5 mends this by introducing *metadata annotations* [4]. On the language level, this is a special kind of class, with limitations on inheritance and types of members, and a peculiar syntax. Figure 3 provides an example based on JSR 175 [4]. Line 7 defines an annotation type `@RFE` with two data fields, one integer and one string; line 10 shows how this annotation type is used. As part of the annotation type definition, we have (on lines 4 and 6) *meta-annotation*, i.e., annotation of the annotation type, that specifies the annotation type properties.

The CLR platform [8], implemented by Microsoft .NET and others, is another example of an environment that recognizes annotations (known here as *custom attributes*) as first-class language constructs all the way from the high-level sources to the runtime environment. There are 23 annotations that are *standard*, i.e., their semantics and format are defined by the platform specification and thus are irrelevant to the discussions in this paper. These annotations carry diverse information, such as runtime-enforced security permissions, CLS compliancy information, native library location, etc. There are also *semi-standard annotations*, i.e., annotations defined by a specific platform implementation (in this case Microsoft .NET). An instrumentation targeting a specific platform implementation may be able to keep up with annotations used by each version targeted, although this may require a significant effort from the instrumentation developers. Also, some of the annotations, not intended for external usage, may be insufficiently documented (this is actually the case for the `@Inheritance` attribute in .NET). Finally, there is a growing body of custom annotations that are defined by individual tools [24]. Instrumentations cannot, in general, learn the semantics and behavior of such annotations.

In the absence of a standard representative collection of annotations that could serve as a benchmark in our discussion, we assembled one of our own (see Table 1). These annotations come from different areas of application, are created by users or tools, targeting online and offline tools, necessary and optional. Some of the papers cited refer to .NET annotations, and some were written before annotations were standardized in either Java or .NET. We modified the original notation (where necessary) to consistently use the Java conventions. These annotations will be used throughout the paper to illustrate our discussion.

3. MODE-BASED CLASSIFICATION

At first glance, annotations present a bewildering multitude of properties and behaviors. For example, correctness of the `@WebMethod` would not be affected by instrumentation that retains the annotation itself, while `@VR` (described in Table 1) could be rendered invalid by any instrumentation of the annotated method. On the other hand, instrumentation must retain `@WebMethod` in order for the application to function correctly, but `@VR` can be removed without causing any harm except perhaps for some performance degradation at first use. Most annotations must be removed if the instrumentation turns them invalid, but some, such as `@StackMap` (cf. Table 1), are easily verified and can be retained even if potentially incorrect. How do we order and group annotations with similar properties?

Ludwig Wittgenstein [35] made the following observation:

> *Imagine a picture representing a boxer in a particular stance. Now, this picture can be used to tell someone how he should stand, should hold himself; or how he should not hold himself; or how a particular man did stand in such-and-such a place; and so on.*

The annotations are similar to that picture. Consider, for example, a field annotated with `@unique` that states the uniqueness of the reference stored in the field. We do not know the meaning of this annotation without its intended usage. It may be used with a runtime that enforces the reference uniqueness by referencing a duplicate object each time a duplicate reference is requested. Alternatively, it may be a request from the user to maintain the uniqueness of the reference, and a conforming runtime would throw an exception if the reference is duplicated. Finally, it may just be indicating that a static analysis tool detected that the program does not duplicate the reference.

3.1 Annotation Modes

There are several ways to encode the annotation usage patterns. We found a simple and elegant approach inspired by the *verbal modes* in grammar. English has three of them: *indicative* for statements of fact ("This reference is unique"), *imperative* for commands ("Reference, be unique!"), and *subjunctive* for wishes and statements that do not necessarily hold ("I request that this reference be unique"). Likewise, we classify annotations according to the following definitions:

Indicative annotations: statements of fact. The information in these annotations can be deduced from the program code. Annotations `@deepImmutableField`, `@array_null_check`, `@VR`, and `@StackMap` are all indicative.

Name	Source	Area	Description
@WebMethod	[22]	Information for compiler and runtime	Marks a method as web method
@ExpectedException	[24]	Information for testing tool	Specifies the expected outcome of the method
@unique	[5]	Component specification	Marks a reference as unaliased
@deepImmutableField	[28]	Program understanding and optimization	Marks a field as immutable
@array_null_check	[30]	Runtime optimization	Marks spurious nullity checks
@VR	[13]	Information for JIT	Virtual register allocation
@StackMap	[34]	Information for JVM	Type mapping of the method stack frame

Table 1: Annotations and their characteristics

Imperative annotations: commands to the runtime or tools, e.g., `@WebMethod`. These annotations cannot be deduced from the program code, and they cannot be violated by anything that the program does.

Subjunctive annotations: requests that may or may not be respected. An action, such as throwing of an exception, may be taken if the request is not satisfied. These annotations cannot be deduced from the program code, but they can be respected or violated by the program. `@ExpectedException` and `@unique` are both subjunctive annotations.

Table 2 classifies the annotations listed in Table 1 according to their usage suggested in the respective papers. Looking at the semi-custom annotations in .NET, our largest real-life source of annotations, we see that `@WebMethod`, together with 218 others, is imperative (the sole possible exception is the undocumented `@Inherited`) [22]. These annotations fall into well-defined groups, each providing for the interaction with a specific tool or part of runtime. Some of the annotations are used at pre-execution stages, such as the component model annotations for visual designers. Others are used by special parts of the runtime, like the interop service annotations that are used for .NET/COM interaction.[4]

Mode	Examples
Imperative	@WebMethod
Subjunctive	@ExpectedException, @unique
Indicative	@deepImmutableField, @array_null_check, @VR, @StackMap

Table 2: Annotation classification

In the literature, imperative annotations appear in Newkirk and Vorontsov's paper on typical annotation usage [24], where they are used to replace marker interfaces and naming conventions, e.g., marking the methods used as test drivers. Subjunctives appear in several papers [24, 16, 14] in the form of pre- and post-conditions and assertions. Since the Java language already contains a runtime-supported mechanism of assertions, it seems that pre- and post-conditions will be the bulk of subjunctive annotations there. Such annotations could be used by a special runtime, or by runtime tools as described in Newkirk and Vorontsov [24], or by a special program transformation tool that would translate these annotations into assertions within the method body. Finally, indicative annotations fall into two large groups. Results of complicated inter-procedural analysis ([28, 7] and others) are normally used by program understanding and development tools. Others, such as those described by Krintz and Calder [18], are used to reduce the overhead of dynamic compilation or to improve performance. Such annotations usually rely on intra-procedural analysis, probably augmented by hierarchy and inheritance information, since in dynamic languages, inter-procedural analyses are rarely reliable.

[4]The standard annotations defined in the CLR Specification are much more diverse; for example, the security permission annotations are subjunctive. However, since they are standard, we are not concerned with them here.

3.2 Modes and Instrumentation

We now consider how annotations belonging to each of these modes behave with respect to the instrumentation. The most striking property of imperative annotations is their insensitivity to instrumentation. Most imperative annotations are only sensitive to changes that involve the removal of the element they are attached to, although some, like `@AutoComplete` in .NET, also require the preservation of some other entities. This is to be expected, since by definition, this is the group of annotations whose correctness cannot be invalidated by normal language means. Therefore, instrumentations, which by definition do not remove existing program elements, can proceed freely in the presence of imperative annotations.

Subjunctive annotations typically modify the semantics, e.g., by throwing an exception or creating a log entry if the requirement that they express is violated. If a subjunctive annotation reacts by exception throwing, it may be possible for the instrumentation to ignore such annotations, provided that the instrumentation catches all the exceptions in the inserted code, even if it does not expect them. In fact, it seems likely that most of the subjunctives belong to this category. However, unless this sub-category is explicitly specified, the only solution that is reasonably safe would be to avoid instrumentation in presence of subjunctive annotations.

Indicative annotations are deducible from the program semantics as it is; therefore, they do not modify it. These annotations are often intended for use by development and static tools, rather than the runtime or runtime tools. Indicative annotations used by the runtime only contain information that the runtime can recompute, and are typically used to take part of the runtime's job offline. Therefore, it is normally safe just to discard these annotations.

These observations could serve as a basis for communications between annotations and instrumentations. The annotation writer, when declaring an annotation type, would describe it (in the same way as annotation targets are described) as being imperative, subjunctive, or indicative. The instrumentation, upon encountering an annotation instance, would examine its type's mode and behave accordingly: ignore if the annotation is imperative, cancel instrumentation if the annotation is subjunctive, and remove annotation if the latter is indicative.

The *mode-based classification* approach has important advantages: it is simple, efficient performance-wise, and intuitive for annotation writers. Sometimes it is not flexible enough, especially for subjunctive annotations, which often

target pre-production tools, and therefore should not prevent instrumentation after the application is deployed. This problem could be alleviated if the annotation writer optionally provided additional information about the annotation, such as the intended lifetime. However, the most significant drawback of mode-based classification is that it is not conservative enough. For example, suppose an indicative annotation is removed by an instrumentation. The tool for which the annotation was intended has all the information needed to re-compute the annotation contents, but it may not have the code for doing this. Without external help, the instrumentation cannot know for sure whether or not its handling of any particular annotation is correct.

4. BEHAVIORAL CLASSIFICATION

In this section, we present a classification scheme that is based on the meta-annotation describing the behavior of the annotation, with the following goals in mind:

- *Precision:* The annotation description should match the annotation behavior closely enough. It must provide good results for "typical" annotations, and even for the outliers we would like to reduce the imprecision.
- *Conservativeness:* Even when the annotation description is not precise, it is important that it enables the detection of all situations where the instrumentation interferes with the annotation, i.e., is conservative. For example, simply ignoring annotation is not sufficient, as demonstrated by Example 1.
- *Ease-of-use:* The classification scheme must avoid requiring significant extra effort from the annotation writer.
- *Scalability:* Performance is crucial for run-time instrumentation; therefore the classification scheme should keep its performance impact as low as possible.

As we have shown in Section 1, the instrumentation cannot determine how to handle annotations in the absence of extra knowledge about the annotations themselves. This knowledge may be conveyed either per annotation instance (attached by the annotation user who created this instance), or per annotation type (attached by the annotation writer who defined this type). Typically, the annotation writer has a better understanding of the annotation's properties. Also, collecting and providing the annotation data per type rather than per instance saves both effort and class file size. Thus we will use a scheme where the annotation writer places the information to be communicated to instrumentation in *meta-annotations*, i.e., annotations of the annotation types, much like `@Target` or `@Retention`.

4.1 Annotation Behavior Characteristics

So what information is necessary? Basically, the instrumentation must know whether its actions affect the annotation correctness and what to do if the answer is positive. The relevant questions here are what, where, how, and when:

- *What?* What changes does the instrumentation apply and what changes is the annotation sensitive to?
- *Where?* Where does the instrumentation apply its changes and where may they affect the annotation?

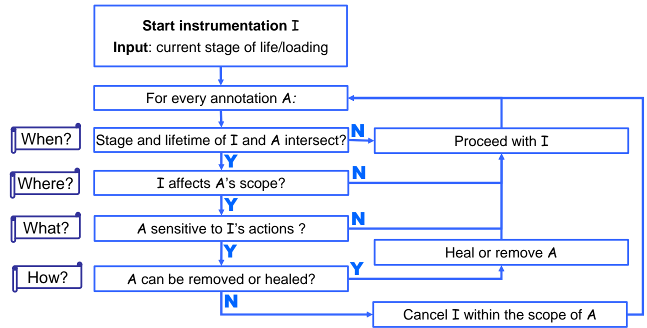

Figure 4: Instrumentation and meta-annotation

- *When?* Is the annotation relevant during the stage of program development and execution when the instrumentation is run?
- *How?* How should the annotation be treated if is (potentially) affected?

Figure 4 outlines how this information could be used at instrumentation time. This is not the only possible approach. For example, extra precision and flexibility could be achieved if the *what/where* questions were addressed to an annotation-declared "arbiter" tool, with instrumentation specification as input. The instrumentation specification, in a conservative way, should be quite feasible, especially if instrumentation is implemented in, say, AspectJ. However, the arbiter tool is extremely problematic — it is a significant burden on the annotation writer, and in all likelihood it would be difficult or impossible to achieve the necessary analysis precision while conforming to the instrumentation's performance requirements. Therefore, for now, it seems necessary to leave the instrumentation in control as in Figure 4, with annotation providing the information.

Of our four questions, *where* and *when* should be easy for the annotation writer to answer. *What* and, to a lesser extent, *how*, are problematic because they require the annotation writer to consider the effects of instrumentation, when the whole goal of instrumentation is to be as unobtrusive as possible. Our mode-based classification approach in Section 3 was intended to avoid such questions, and it is insufficient — for a reason. One must either limit the annotations' genericity and require them to behave according to the rules set for their respective category (e.g., require that tools using indicative annotations function correctly in the absence of these annotations), or else require the annotations to provide specific instructions to the instrumentation, in which case annotation can remain generic. We chose the latter approach. Below, we discuss the annotation description in more detail. We also discuss annotation attachment information and why it is important for instrumentation.

4.2 Where and What: Scope and Sensitivity

The *annotation scope* is the set of classes and class elements whose modification may affect the correctness of an annotation. For example, an annotation is package-scoped if only changes to classes within the package may render it incorrect. Some annotations may have scopes that are quite difficult to describe. For example, a security annotation claiming that a method only accesses files within a certain directory actually covers not only the method itself,

but also its direct and indirect callees, except those that exclude themselves.

Given a scope, *annotation sensitivity* describes which changes within the annotation scope can render the annotation incorrect. For example, `@WebMethod` is not sensitive to instrumentation changes within the program; `@array_bounds_check` is sensitive to any change within the annotated method's body, because instrumentation moves instructions to different offsets (see also Section 4.5). Generally, an annotation can be described using a set of (scope, sensitivity) pairs; but for all the annotations we have examined, one such pair sufficed.

The only way to conservatively describe the annotation sensitivity is to declare it sensitive to any changes. We choose this option as default, but allow the annotation writer to relax it (cf. Table 3). Creating a vocabulary for describing annotation scope is also not straightforward. Many annotations, e.g., `@unique`, have *dynamic* scopes such as "all the methods that may access the annotated field, together with their callees", and these scopes are difficult to describe statically. Even if such a description were added to our language, using it may impose too high performance penalties on instrumentations. Further complexities in Java are added by its flexible class loading. While at the Java source code level it makes sense to talk about application- or package-wide scope, in bytecode such annotations can only be supported using a special-purpose annotation type to aggregate the restrictions at the package or application levels. Such an annotation should be placed by the compiler as a package-level bytecode annotation (for package-scoped sensitivity) or at every application entry point (for application-level sensitivity). With this provision, it is possible to comply with the sensitivity language as defined in Table 4. The possible scopes are similar to their corresponding Java access modifiers, except for `CURRENT`, which improves scope definition for annotations attached to entities that are not class members, e.g., parameters or method bodies. An alternative solution would be to distinguish between annotations referring to declarations of classes and methods and those referring to their implementations.

Name	Description
`ANY_CHANGE`	Annotations sensitive to all changes within the scope (default)
`NON_SPECTATIVE_CHANGE`	Annotations not sensitive to spectative changes (cf. Section 2.1) within the scope

Table 3: Annotation sensitivity

4.3 How: Annotation Healing

Rendering an annotation invalid should not necessarily prevent instrumentation. Indeed, most of the annotations suggested in the literature, such as `@array_bounds_check`, have an advisory character. The healing meta-annotation serves to mark an annotation whose absence can be fixed by tools downstream. At the very least, we'd like to know if the annotation is optional or mandatory. Other annotations may convey information that is much cheaper to check than to re-generate (e.g., `@StackMap`); such annotations may be retained even if violated. The proposed healing options are enumerated Table 5. For example, healing for `@WebMethod` is `CANCEL`, while for `@null_array_checks` it is `REMOVE`.

Name	Description
`CURRENT`	Annotations sensitive only to changes to the program element (e.g., class, method, parameter) to which they are attached
`MEMBER`	Annotations sensitive only to changes to the class members to which or to whose sub-elements they are attached
`CLASS`	Annotations sensitive only to changes to the classes to which or to whose sub-elements they are attached
`HIERARCHY`	Annotations sensitive only to changes to the classes to which or to whose sub-elements they are attached, and to their subtypes
`PACKAGE`	Annotations sensitive only to changes to the classes to which or to whose sub-elements they are attached, and other classes in the same package (default)
`PROGRAM`	Annotations sensitive to changes anywhere within the program

Table 4: Annotation sensitivity scope

Name	Description
`CANCEL`	Annotations must be preserved, so changes must be canceled (default)
`IGNORE`	Annotations can be retained even if invalid
`IGNORE_CATCH`	Annotations can be retained even if invalid, but this may result in unexpected exceptions in the modified code
`REMOVE`	Annotations must be removed

Table 5: Annotation healing

4.4 When: Annotation Lifecycle

While annotation scope seeks to separate annotation and instrumentation spatially, the properties that describe annotation usage attempt to separate them in time. For example, in Java, `@Retention` meta-annotation (possible values: `SOURCE`, `CLASS`, `RUNTIME`) indicates whether the annotation should be preserved in the source, class file, or at runtime respectively. We augment `@Retention` by *lifecycle* information (cf. Table 6) to get a better picture of the annotation usage. An additional benefit is that such markup can help ensure that this sensitive annotation is removed from the binaries before shipping.[5] For example, the lifecycle of `@WebMethod` and `@null_array_checks` is `DEPLOYMENT`; the lifecycle of `@ExpectedException`, used by a testing suite to mark the test outcome, is `INTEGRATION`.

Name	Description
`DEVELOPMENT`	Annotations are last used during component development and testing
`INTEGRATION`	Annotations are last used during the development and testing of components that use the annotated component
`DEPLOYMENT`	Annotations are last used at the component production stage (default)

Table 6: Annotation lifecycle

4.5 Annotation Attachment

Every annotation comes attached to some program element. The importance of this information stems from the fact that it facilitates instrumentation by limiting the anno-

[5]The lifecycle can be tailored to a specific software process lifecycle, such as Rational Unified Process.

tation scope. Consider, for example, an annotation providing an intra-procedural register allocation scheme. It is extremely volatile; intrusive instrumentation can either leave the method alone or remove the annotation. However, an external instrumentation could have moved the method body (together with the annotation attached to it) to a new method and so achieved its purpose while retaining the optimizing annotation.

Another example is `@array_null_check`, which describes an individual bytecode instruction. Assume that a certain type of annotation is attached to instructions whenever an analysis establishes that the usual reference nullity checks can safely be omitted for these instructions. Since currently there is no way to annotate individual instructions, this would be implemented as a method-level annotation containing the instruction offset as a data field. An intrusive instrumentation of the annotated method would make such annotations unsafe, since the instruction offsets are changed due to instrumentation and the annotations would refer to incorrect instructions. Therefore, the annotation writer should declare this annotation as sensitive to all changes. If there were a way for annotation to be attached to an individual instruction (i.e., if the annotation standard would make explicit the offset semantics of the annotation's data field), the instrumentation would receive the responsibility for fixing this field (much as line number tables are fixed), and again the annotation could be retained in spite of instrumentation.[6]

At the source code level, both Java and C# allow attaching annotation to parts of the method, e.g., individual parameters. However, neither language distinguishes between annotations attached to method declarations and method bodies, and neither persists the sub-method attachment information to the bytecode level. We propose to add, both to the high-level languages and the underlying platforms, a means to specify the entity to which the annotation is attached. Without this information, the instrumentation would be forced to be much more conservative than necessary.

4.6 Practical Experience

In this section, we evaluate how the proposed solution fares according to the criteria outlined in the beginning of Section 4. Our descriptions are based on a vocabulary of simple terms, which is easy for the annotation writer to use, can be efficiently consumed by the instrumentation, and allows a conservative choice of default values. The scope specification is the one that suffers the most from the simplicity of our descriptions, especially for annotations whose scope is a runtime property (such as all callees of a given method). However, if the language for the scope is more complicated, it becomes difficult for the instrumentation to identify whether or not its modifications affect the scope of the annotation. Therefore, we sacrifice precision for the sake of scalability.

Our research on the interaction between annotations and instrumentation was motivated by our work on a load-time instrumentation intended to become a part of a large industrial product that monitors programs after their deployment. Therefore, in order to evaluate our approach, we turn to this motivating example and examine how it would be able to proceed in the presence of the the annotations in Table 1. Figure 5 presents their meta-annotation.

Note how the change in attachment information affects the sensitivity (though not the scope!) of `@array_null_check`. The actual scope for `@WebMethod` and `@ExpectedException` would be the program slice rooted in the annotated method rather than the whole program. Likewise, for field annotations `@unique` and `@deepImmutableField`, the scope depends on escape properties of the annotated field (in the latter case also all variables reachable from it), and could be significantly narrower than the the whole program. However, the precise scope in all those cases is extremely difficult to define and describe statically, and `PROGRAM` scope seems a reasonable choice. For the rest, the scope annotation is pretty close to the sensitivity scope. Note also the `@StackMap` annotation, which is costly to compute but cheap to verify, so it can be retained even if there is a risk that it was rendered invalid. Table 7 shows how each of the annotations interacts with our sample instrumentation. In particular, we see that all the critical annotations can be retained without risk using our scheme.

Name	Instrumentation impact
`@WebMethod`	None (not sensitive)
`@ExpectedException`	None (not sensitive, different lifecycle)
`@unique`	None (different lifecycle; could be removed)
`@deepImmutableField`	None (insensitive)
`@array_null_check`	None if instruction-attached (not sensitive), remove annotation if method-attached
`@VR`	Remove annotation
`@StackMap`	None, even if affected

Table 7: Annotations and the sample instrumentation

5. RELATED WORK

We have shown that one cannot trivially instrument an annotated programs without affecting the program semantics of the resulting program. The problem of the interference between program annotation and instrumentation is not benign. To the best of our knowledge this problem has not been addressed before. It is related to the *independent extensibility* problem in aspect-oriented programming. Indeed, instrumentations can often be implemented as aspects (e.g., [6]), and annotations can also be seen as a transformation of the original problem. The independent extensibility problem is still open except for the special case of transformations that only affect the interfaces [15]. In this paper, we are dealing with a special case of two transformations, A (the annotation) and I (the instrumentation), where I operates on the code where A is present as a definition but potentially was not yet applied. There remains ambiguity in the interaction of different annotations at runtime, which is a manifestation of the general extensibility problem. There have also been numerous works on correctness (i.e., semantics-preservation) of compiler transformations, for synchronous [29] and imperative [31][19] languages. However, this approach does not work for annota-

[6] An alternative solution would be to define, for each method, a table that maps numerical labels to method offsets, similar to the existing line number table. The instrumentation would be to fix this table, again similar to what is done for line number tables, and the dependence of annotations on offset values would be reduced.

```
@Target(METHOD} @Retention(RUNTIME) @Lifecycle(DEPLOYMENT)
@Sensitivity(NON_SPECTATIVE_CHANGE) @SensitivityScope(PROGRAM) @Healing(CANCEL)
public @interface WebMethod{...}

@Target(METHOD} @Retention(RUNTIME) @Lifecycle(INTEGRATION)
@Sensitivity(NON_SPECTATIVE_CHANGE) @SensitivityScope(PROGRAM) @Healing(CANCEL)
public @interface ExpectedException{...}

@Target(FIELD} @Retention(CLASS) @Lifecycle(INTEGRATION)
@Sensitivity(ANY_CHANGE) @SensitivityScope(PROGRAM) @Healing(CANCEL)
public @interface unique{...}

/* deepImmutableField as defined for program optimization */
@Target(FIELD} @Retention(RUNTIME) @Lifecycle(DEPLOYMENT)
@Sensitivity(NON_SPECTATIVE_CHANGE) @SensitivityScope(PROGRAM) @Healing(REMOVE)
public @interface deepImmutableField{...}

/* deepImmutableField as defined for program understanding */
@Target(FIELD} @Retention(CLASS) @Lifecycle(INTEGRATION)
@Sensitivity(NON_SPECTATIVE_CHANGE) @SensitivityScope(PROGRAM) @Healing(REMOVE)
public @interface deepImmutableField{...}

/* array_null_check attached to instruction (currently not available) */
@Target(INSTRUCTION} @Retention(RUNTIME) @Lifecycle(DEPLOYMENT)
@Sensitivity(NON_SPECTATIVE_CHANGE) @SensitivityScope(ELEMENT) @Healing(REMOVE)
public @interface array_null_check{...}

/* array_null_check attached to method */
@Target(METHOD} @Retention(RUNTIME) @Lifecycle(DEPLOYMENT)
@Sensitivity(ANY_CHANGE) @SensitivityScope(ELEMENT) @Healing(REMOVE)
public @interface array_null_check{...}

@Target(METHOD} @Retention(RUNTIME) @Lifecycle(DEPLOYMENT)
@Sensitivity(ANY_CHANGE) @SensitivityScope(ELEMENT) @Healing(REMOVE)
public @interface VR{...}

@Target(METHOD} @Retention(RUNTIME) @Lifecycle(DEPLOYMENT)
@Sensitivity(ANY_CHANGE) @SensitivityScope(ELEMENT) @Healing(IGNORE)
public @interface StackMap{...}
```

Figure 5: Annotations and their meta-annotations

tions, whose semantics are not known to the translator or compiler.

Another related area is *unintended software evolution*, which mostly focuses on refactoring transformations. Refactoring transformations aim to enhance the structure of evolving software, for example, by outlining common code. This differs from the instrumentations we discuss in this paper, which typically do not affect the program structure. The usage scenario is also different; while refactoring is typically performed on the source code by a developer, instrumentations are typically performed on binaries or class files by a user.

Some of the numerous works on instrumentations and annotations are mentioned in Section 2. Krintz and Calder [18] address the problem of class file corruption that causes invalid annotations, and distinguish between annotations that are safe to use (probably at the price of performance) even if invalid, and those that are unsafe and thus must be validated if used. They suggested using only annotations that do not affect program correctness or verifying the annotations at runtime. This may be a workable solution for the indicative annotations they worked with, but not with imperative or subjunctive ones that cannot be verified. However, instead of a malicious corruption we are dealing with instrumentation, which is a cooperating agent. More recently, Krintz [17] presented a technique that substantially improves Java program performance by incorporating both on-line and off-line profile information to guide dynamic optimization. The information is passed using bytecode annotation. Although annotations are used in conjunction with instrumentation to guide dynamic optimization, the interference that could happen between instrumentation and annotation was orthogonal to the problem addressed by Krintz [17].

Pechtchanski [27] proposes a different annotation taxonomy, based on the annotation's effect on program correctness and whether it is verifiable. While the verifiability criterion is common to both taxonomies, the correctness effect is replaced in our paper with deducibility. One reason is that it is sometimes difficult to say whether a specific annotation affects correctness or not. For example, it is not obvious whether correctness is modified by an annotation that provides textual description of a GUI component, but it is obvious that such description could not be deduced from a program analysis. Alternatively, the `@WebMethod` annotation adds new behaviors but preserves the old ones, so does it affect correctness or not? Also, some annotations are only used at certain project lifecycle stages (e.g., an annotation that indicates the expected method outcome for a testing tool). So while the information conveyed by them affects the tool's behavior, it does not modify the semantics of the program itself, and the program may actually be expected to run normally without this tool.

Another related direction of research is impact analysis [32]. Advances in this area could let us capitalize on the relative simplicity of describing instrumentations as opposed to describing generic annotations.

6. CONCLUSIONS AND FUTURE WORK

Instrumentation is a very powerful mechanism for collecting program feedback and runtime profile information, and is widely used both in industry and in research. Annotations have been used in a variety of research projects to pass information between programmers, tools, and runtime environments. Now they are making their way into the industrial mainstream, through their wide usage in CLR and their recent standardization in Java. This paper explores the previously overlooked problem of maintaining the correct semantics and behavior of annotated programs after their instrumentation. We show that without active cooperation between the two, instrumentation of annotated code is unsafe.

We propose a taxonomy of annotations based on our study of more than two hundred annotations that appear in literature or are used on .NET and Java platforms. We use this taxonomy to derive a classification of annotations with respect to instrumentation. We show how meta-annotation describing instrumentation-relevant annotation properties can protect against using corrupted annotations, and demonstrate this solution on a set of sample annotations drawn both from industrial and research usages of annotations.

There is much work remaining for further research. While we have focused on how instrumentation can interfere with annotations, the interference can be the other way round as well. For example, it is important, in profiling and many other contexts, that the instrumentation cover all the entry points into the software component. Some annotations, such as `@WebMethod`, turn a usual method into a component entry point in certain environments. Such information is very important for instrumentation. One could also consider other types of program transformations. We think it is very important to bring these issues to the forefront while the usage of annotations is in its infancy and there is time to apply the necessary changes.

Acknowledgements

We would like to thank Gad Haber, Michael Hind, Igor Pechtchanski, Erez Petrank, Peter Sweeney, and the anonymous referees for their helpful comments on this work.

7. REFERENCES

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In OOPSLA 2002 [25], pages 311–330.

[2] A. Azevedo, A. Nicolau, and J. Hummel. An annotation-aware Java virtual machine implementation. *Concurrency: Practice and Experience*, 12(6):423–444, 2000.

[3] C. Baquero and F. Moura. Concurrency annotations in C++. *SIGPLAN Notices*, 29(7):61–67, 1994.

[4] J. Bloch. JSR 175: A metadata facility for the Java programming language. http://jcp.org/en/jsr/detail?id=175.

[5] E. C. Chan, J. T. Boyland, and W. L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the 20th International Conference on Software Engineering*, pages 167–176. IEEE Computer Society, 1998.

[6] M. Debusmann and K. Geihs. Efficient and transparent instrumentation of application components using an aspect-oriented approach. In *14th IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM 2003)*, Heidelberg, Germany, October 2003.

[7] D. Detlefs, K. R. M. Leino, G. Nelson, and J. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, CA, Dec. 1998.

[8] ECMA. *Standard ECMA-335. Common Language Infrastructure (CLI)*. ECMA, 2nd edition, December 2002.

[9] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002. Also available as `http://www.research.ibm.com/journal/sj/411/-edelstein.html`.

[10] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[11] R. Field. JSR 163: Java platform profiling architecture. http://jcp.org/en/jsr/detail?id=163.

[12] A. Goldberg and K. Havelund. Instrumentation of Java bytecode for runtime analysis. In *Proceedings of the ECOOP 2003 Workshop on Formal Techniques for Java-like Programs, Technical Report 408*, 2003.

[13] J. Jones and S. N. Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):423–444, 2000.

[14] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In OOPSLA 2002 [25], pages 231–245.

[15] G. Kniesel, P. Costanza, and M. Austermann. Independent extensibility for aspect-oriented systems. In *ASC Workshop, ECOOP 2001*, Budapest, Hungary, 2001.

[16] B. Krieg-Brückner and D. C. Luckham. ANNA: Towards a language for annotating Ada programs. In *Proceeding of the ACM-SIGPLAN Symposium on Ada Programming Language*, pages 128–138. ACM Press, 1980.

[17] C. Krintz. Coupling on-line and off-line profile information to improve program performance. In *International Symposium on Code Generation and Optimization (CGO03)*, 2003.

[18] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI)*, pages 156–167, Snowbird, Utah, June 2001. ACM SIGPLAN, ACM Press. SIGPLAN Notices 36(5).

[19] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 220–231. ACM Press, 2003.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[21] K.-P. Löhr. Concurrency annotations. In A. Paepcke, editor, *Proceedings of the 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 327–340, Vancouver, British Columbia, Canada, Oct.18-22 1992. OOPSLA'92, ACM SIGPLAN 27(10) Oct. 1992.

[22] Microsoft. The .NET framework class library. http://msdn.microsoft.com/library/default.asp?url=/library/-en-us/cpref/html/cpref_start.asp.

[23] N. H. Minsky. Towards alias-free pointers. In P. Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming*, number 1098 in Lecture Notes in Computer Science, Linz, Austria, July 1996. ECOOP'96, Springer Verlag.

[24] J. Newkirk and A. A. Vorontsov. Design: How .net's custom attributes affect design. *IEEE Softw.*, 19(5):18–20, 2002.

[25] OOPSLA 2002. *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Seattle, Washington, Nov. 4–8 2002. ACM SIGPLAN 36(11) Nov. 2002.

[26] Open Group. *Technical Standard C014: Application Response Measurement Issue 3.0 Java Binding*. Open Group, October 2001.

[27] I. Pechtchanski. *A Framework for Optimistic Program Optimization*. PhD thesis, New York University, New York, NY, 2003.

[28] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *The Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 202–211. ACM Press, November 2002.

[29] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–??, 1998.

[30] F. Qian, L. J. Hendren, and C. Verbrugge. A comprehensive approach to array bounds check elimination for Java. In R. Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 325–342. Springer, 2001.

[31] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[32] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53. ACM Press, 2001.

[33] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003.

[34] A. Taivalsaari. JSR 139: Connected limited device configuration 1.1. http://jcp.org/en/jsr/detail?id=139.

[35] L. Wittgenstein. *Philosophische Untersuchungen — Philosophical Investigations*, chapter I.22. Blackwell, Oxford, 1953.