Checking Spring Annotations

Konrad Fögen, Vincent von Hof, and Herbert Kuchen

University of Münster, IS Department, Leonardo Campus 3, D-48149 Münster,
konrad.foegen@uni-muenster.de, von.hof@wi.uni-muenster.de,
kuchen@uni-muenster.de,
WWW: https://www.wi.uni-muenster.de/department/groups/pi/

Abstract. Dependency injection frameworks such as the Spring framework rely on dynamic language features of Java. Errors arising from the improper usage of these features bypass the compile-time checks of the Java compiler. This paper discusses the application of static code analysis as a means to restore compile-time checking for Spring-related configuration errors. First, possible errors in the configuration of Spring are identified and classified. Attributed grammars are applied in order to formally detect the errors and a prototypical compiler extension is implemented based on Java's pluggable annotation processing API.

1 Introduction

The Java programming language is one of the most popular programming languages in general and especially in the field of enterprise applications[1] [Wal14, p.3]. In addition, *dependency injection* (DI) is frequently used to support and simplify the development of Java applications. DI is a *creational design pattern* that abstracts away the process of object creation and composition [GHJV95, p.94]. Typical implementations are generic and make no assumptions about the objects they manage. Instead, they rely on an external configuration [Pra09, p.17].

However, the generic implementation requires the use of dynamic language features such as the *Java Reflection API* [Ora15]. Despite its necessity and usefulness, the Java Reflection API has a downside, since errors arising from the improper application cannot be detected by available Java compilers. Thereby, the detection of errors is shifted from compile-time to runtime.

Since the errors are not automatically detected at compile-time and the manual detection is tedious, developers have a particular interest in automatic solutions to detect them as early as possible, preferably at compile-time during the development.

There are quite a few tools for the *static analysis* of Java programs such as FindBugs [APM+07], Checkstyle [Bur03], PMD [PMD15], SonarQube

[1] See http://www.langpop.com or http://lang-index.sourceforge.net/ as indicators for Java's popularity.

		Spring Context	
		Dependent	**Independent**
Analysis Level	**Above Method Level**	Group I (13 errors)	Group II (13 errors)
	Below Method Level	Group III (8 errors)	Group IV (4 errors)

Table 1. Classification of Spring Configuration Errors

[Son15], Java Language Extender [VWKBJ06], ESC/Java2 [CK05], JastAddJ [EH07], JavaCOP [MME+10], JQual [GF07], and the Checker framework [PAC+08]. All these tools are general purpose inspection tools. To the best of our knowledge, there is no tool for the static detection of Spring configuration errors.

Our approach is based on attributed grammars [Knu68] and a compiler extension to detect errors arising from the improper application of the *Spring framework* [Piv15] at compile-time. Spring is chosen as a representative for the multitude of different DI implementations for Java, since it is one of the most popular implementations. Furthermore, its configuration is based on Java annotations which makes it very suitable for pluggable annotation processing.

This paper is structured as follows. In Section 2, Spring configuration errors are identified and classified. In Section 3, attributed grammars are provided which formally describe the detection of Spring configuration errors. Using these attributed grammars, a prototypical compiler extension based on Java's pluggable annotation processing API is described in Section 4. The insights gained by the development of the prototype are used to evaluate the approach in Section 5. In Section 6, we conclude and point out future work.

2 Spring Configuration Errors

The Spring framework is an open source framework which implements the dependency injection design pattern. At its core, the *Spring context* component provides its clients with requested and dependent objects, so-called *beans* which can be any kind of simple, Plain Old Java Objects (POJOs) [Wal14, p.4].

Several different context implementations exist which mainly differ in terms of their configuration format, e.g. Java-based or XML-based configurations. In general, configurations consist of features referring to the actual *Spring context* as well as to a set of Spring bean definitions.[2] Besides that, three different ways of defining *Spring beans* are supported by Spring: explicit configurations via Java and XML as well as implicit configurations via Java annotations. For more information about Spring, please refer to [Wal14], [Pra09] or [Joh15].

A literature review of the Spring framework reference [Joh15] and expert interviews have been conducted in order to identify different types of errors. As a result, 38 distinct error types have been identified and classified into four groups as depicted in Table 1. In the present paper, the core container and the

[2] This work focuses only on Java-based configurations.

data access / integration modules are considered, since they can be used by any Spring-based Java application. The classification depends on the core features of the Spring framework and the usage of these features determines the assignment of an error to a certain dimension. Four groups are formed by two dimensions each featuring two manifestations. The spring context dimension determines whether or not the analysis requires information derived from the Spring context. Analysis level is the second dimension and determines whether or not the analysis requires information about the control flow that concerns language constructs below the method-level, e.g. method invocations or variable assignments. In the following, exemplary errors are described for further illustration of the different error types.

Group I. Errors belonging to this group depend on the Spring context and another component which uses a related Spring-specific annotation. The errors occur *above* the method level, e.g. declarations of classes, methods or member variables. Viewed in isolation, the Spring context configuration and the component may not be erroneous but their interaction is.

As an example, Spring's transaction infrastructure encapsulates the internals of specific transaction management APIs and offers a declarative model for the integration into applications [Joh15, chap.12.3]. A Spring context which defines a transaction manager as well as a *@EnableTransactionManagement* annotation are required to enable the transaction management. Once enabled, the *@Transactional* annotation can be attached to methods in order to enable transactional support for the method. Listing 1.1 illustrates the correct usage.

```
1  @Configuration
2  @EnableTransactionManagement
3  public class SpringConfig {
4    @Bean
5    public PlatformTransactionManager transactionmanager() {
         ...}
6  }
7
8  @Component
9  public class PrinterService {
10   @Transactional
11   public void print() {...}
12 }
```

Listing 1.1. Illustration of Spring's Transaction Management

In this situation, errors occur if transaction management is enabled but not used because no methods are annotated with *@Transactional*, i.e. when line 10 is removed from the listing. Or - the opposite situation - if *@Transactional* methods exist but the transaction management is not enabled, i.e. a situation where line 2 is removed from the listing.

Group II. Errors in group II are Spring context-independent and occur above the method level. They consist of annotated language constructs which - at the same

time - have some other attributes or other annotations which are incompatible to that original annotation.

For instance, the Spring framework uses the *@Autowired* annotation to mark methods or fields for annotation-based injection [Wal14, p.39]. Sometimes, dependencies are ambiguous and the Spring context finds several bean definitions that match and then has to choose between them [Wal14, p.75]. The *@Qualifier* annotation can be used in conjunction with *@Autowired* to narrow the result set. It is an error to use *@Qualifier* without a corresponding *@Autowired* annotation.

The *@Qualifier* annotation can also be used indirectly. There is no difference between the usage of *@Qualifier* and the usage of annotation types annotated with *@Qualifier*. Both, the error and the indirect usage of *@Qualifier* are illustrated in Listing 1.2.

```
1  @Qualifier
2  @interface DinA4Format {...}
3
4  @Component
5  @DinA4Format
6  class DinA4DocumentFormatter implements DocumentFormatter {
       ...}
7
8  @Component
9  class PrinterService {
10   //@Autowired is missing
11   @DinA4Format
12   public PrinterService(DocFormatter f) {...}
13 }
```

Listing 1.2. Illustration of @Qualifier Without @Autowired

Group III. Similar to errors in group I, the errors in this group also depend on specific Spring context configurations. But in contrast, they also depend on language constructs *below* the method level. For instance, the lifecycle of a bean is an important aspect described by its *bean definition*. The lifecycle of a bean starts after the corresponding Spring context is initialized and ends right before the Spring context shuts down. In between that timeframe, the lifecycle of a bean is defined by its *scope* [Wal14, p.81]. The *singleton* scope is the default scope for beans, where only one shared instance of the bean exists per Spring context and it exists until the context shuts down [Joh15, chap.5.5.1]. In contrast, beans defined with the *prototype* scope are created as new instances every time they are requested.

One important aspect regarding prototype scope is that the Spring context does not manage the complete lifecycle of these beans. Even though prototype and singleton beans are initialized the same way, their destruction is different, since the Spring context does not store references to prototyped beans and therefore cannot initiate their destruction. A Spring bean definition is explicitly defined by attaching the *@Bean* annotation to a method or implicitly defined by adding the *@Component* annotation to a class declaration.

Furthermore, Spring allows to define *lifecycle callbacks*, e.g. methods called by the Spring context when a bean is constructed or destructed [Wal14, p.33]. Among other ways, they can be defined by annotating a corresponding method with *@PostConstruct* or *@PreDestroy*.

Since the Spring context cannot initiate the destruction process of prototyped beans, methods that qualify as destruction lifecycle callbacks are considered erroneous, if the beans are defined with the prototype scope. Listing 1.3 illustrates this error where the component contains a *close* method to cleanup resources which is never invoked due to the prototype scope.

```
1  @Configuration
2  public class SpringConfig {
3    @Bean
4    @Scope("prototype")
5    public PrinterService printerService() {
6      return new PrinterService();
7    }
8  }
9
10 public class PrinterService {
11   @PreDestroy
12   public void close() {   // will not be invoked
13     this.usbConnection.close();
14   }
15 }
```

Listing 1.3. Illustration of Callbacks on Prototyped Beans

Group IV. This group also comprises errors that occur below the method level and do not depend on the Spring context. An example is related to Spring's *JdbcTemplate* component which provides an abstraction layer covering specific details of Java's JDBC API. Here, SQL is used to interact with databases. The corresponding code is *linguistically separated* from the surrounding code written in Java. Therefore, the compiler cannot check whether or not SQL code is compliant to the language definition of SQL. A typical use case in this context is a developer who creates and tests SQL statements in a database tool. Once the SQL statement is ready, he copies it into a Java String and uses it. Statements which are executed in a database tool often require to be terminated with a semicolon. However, having that semicolon at the end of a SQL String in Java results in a runtime exception.

These kinds of problems can be detected by applying pluggable type systems similar to the one for regular expressions by the Checker framework [SDE12]. Therefore, such errors are not further discussed in this paper.

3 Error Detection via Attributed Grammars

Static code analysis is an analytical approach to detect lexical, syntactic, and also some semantic errors. The source code of software is analyzed in order to

understand its structure (syntax) and meaning (semantics) [ALSU07, p.21]. The gained understanding is then used to identify errors within the source code. Regular expressions and context-free grammars can be used to define the lexical and syntactic structure of a programming language and errors can be detected by identifying mismatches between the defined structure and the actual source code.

Beyond syntactical checks, compilers may also check for the static semantics, e.g. whether a method is invoked with the right number and types of arguments. Or, in our case, that Spring configurations and annotations are used in a correct manner.

In 1968, KNUTH introduced *attributed grammars* as a formal approach to express and handle semantical aspects of a programming language [Knu68]. Attributed grammars are context-free grammars extended with attributes and semantic rules [SK95, pp.66-67]. Each nonterminal of the context-free grammar may have several attributes. Each attribute can either be *synthesized* or *inherited* and it has a value which is defined by a semantic rule associated with a production of the context-free grammar.

Roughly, if $A ::= B_1 \dots B_n$ (for $n \in \mathbb{N}$) is a context-free rule with nonterminals $A, B_1, \dots, B_n$, all of which have a synthesized attribute s and an inherited attribute i, then corresponding semantic rules can define the values of the attributes $A.s, B_1.i, \dots, B_n.i$ as follows:

$$A.s \leftarrow f(A.i, B_1.s, \dots, B_n.s) \tag{1}$$

$$B_j.i \leftarrow g(A.i, B_1.s, \dots, B_{j-1}.s, B_{j+1}.s, \dots, B_n.s) \tag{2}$$

where f and g are functions mapping attribute values to another attribute value and $j \in \{1, \dots, n\}$. If the context-free rules contain terminal symbols and / or nonterminals have several synthesized and inherited attributes, the formulas (1) and (2) have to be generalized accordingly (see [ALSU07] for a full description of attributed grammars).

The following notation is used within this paper to represent attributes, semantic rules, and conditions. Semantic rules are enclosed by curly brackets and are placed behind the body of the corresponding production. $\$0.a$ is used to refer to attribute a of the symbol on the left hand side (lhs) of the production, $\$1.a$ is used to refer to the leftmost symbol of the right hand side (rhs) and so forth. A reversed arrow $\leftarrow$ is used to represent the value assignment from the value on the rhs of a semantic rule to the attribute on the lhs. For the rhs, we use a syntax similar to that of C or Java.

After constructing a syntax tree (via lexical and syntactic analysis), the attribute values of the symbols in that tree can be determined by applying the semantic rules [ALSU07, p.54]. The order in which the attributes can be evaluated has to reflect the dependencies of the attributes caused by the semantic rules. In general, there is no guarantee that an order exists in which all attributes of all nodes can be evaluated. Though, there are subclasses of attributed grammars which restrict the usage of attributes and semantic rules to guarantee the existence of an evaluation order [ALSU07, p.313].

⟨*root*⟩ ::= ⟨*typedecllist*⟩ | $

⟨*typedecllist*⟩ ::= ⟨*typedecl*⟩ ⟨*typedecllist*⟩ | ε

⟨*typedecl*⟩ ::= ⟨*modifiers*⟩ ⟨*typedecltype*⟩

⟨*modifiers*⟩ ::= '`public`' ⟨*modifiers*⟩ | '`private`' ⟨*modifiers*⟩
  | ⟨*annotation*⟩ ⟨*modifiers*⟩ | ε

⟨*annotation*⟩ ::= '`@`' ⟨*identifier*⟩ ⟨*annoarguments*⟩

⟨*typedecltype*⟩ ::= ⟨*annotypedecl*⟩ | ⟨*classdecl*⟩ | ⟨*interfacedecl*⟩

⟨*annotypedecl*⟩ ::= '`@interface`' ⟨*identifier*⟩ '{' ⟨*annotypebody*⟩ '}'

⟨*classdecl*⟩ ::= '`class`' ⟨*identifier*⟩ ⟨*superclass*⟩ ⟨*interfaces*⟩ '{' ⟨*classbody*⟩ '}'

⟨*interfacedecl*⟩ ::= '`interface`' ⟨*identifier*⟩ ⟨*superinterface*⟩ '{' ⟨*interfacebody*⟩ '}'

⟨*classbody*⟩ ::= ⟨*modifiers*⟩ ⟨*type*⟩ ⟨*identifier*⟩ ⟨*classbodytype*⟩ ⟨*classbody*⟩ | ε

Fig. 1. Java Grammar in BNF Notation (Excerpt).

Two subclasses relevant for this work are *S-* and *L-attributed grammars*: S-attributed grammars are grammars that only contain synthesized attributes and no inherited attributes [ALSU07, p.313]. They allow a bottom-up evaluation of attributes. L-attributed grammars also guarantee the existence of an evaluation order. Roughly, they allow the evaluation of attributes bottom up and left to right. See [ALSU07] for details.

We use an LL(1)-compliant context-free grammar which describes the subset of Java language constructs relevant for the detection of Spring configuration errors. Figure 1 provides an overview of the productions which are relevant for the succeeding analyses.

The semantic rules used for attributed grammars are based on the following constants and operations: *error* is used to indicate that an error has been detected, *emptySet* creates an empty set, *newSet* creates a set containing a single element, *intersects* returns true if and only if an intersection of two sets is not empty and *union* computes the union of two sets. The function *value* operates on identifiers and returns the actual value as a string.

In the following, an exemplary attributed grammars is provided which allow to expose an error explained in of Section 2.

Spring context-dependent errors that occur above the method level can be generically depend on the presence or absence of annotations. The presence or absence can be described via two synthesized boolean attributes *enabled* (1) and *used* (2). A boolean expression using the two attributes describes whether an error is present or not.

Consider again the error introduced in Listing 1.1. It can be exposed as follows. The attribute *enabled* is set to true, if a Spring configuration exists and if this configuration is annotated with *@EnableTransactionManagement*. The

Nonterminal Symbols	Synthesized Attributes
$\langle root \rangle$	-
$\langle typedecllist \rangle$	enabled, used
$\langle typedecl \rangle$	enabled, used
$\langle typedecltype \rangle$	used
$\langle classdecl \rangle$	used
$\langle classbody \rangle$	used
$\langle interfacedecl \rangle$	used
$\langle interfacebody \rangle$	used
$\langle modifiers \rangle$	names
$\langle annotation \rangle$	name

Table 2. Overview of Nonterminals and Synthesized Attributes.

attribute *used* is set to true, if a method annotated with *@Transactional* exists. Otherwise, the attributes are set to false. The error occurs if the attribute *enabled* is true and at the same time the attribute *used* is false. The condition can thus be expressed as the boolean expression $enabled \wedge \neg used$.

The detection can be described by an S-attributed grammar with four synthesized attributes *enabled*, *used*, *name* and *names* whereby *enabled* refers to occurrences of *@EnableTransactionManagement* and *used* refers to occurrences of *@Transactional*. *name* refers to the identifier of an annotation and *names* is a set of a names.

In the following, the semantic rules and the synthesis are described in greater detail. Table 2 provides an overview of the relevant nonterminals and their synthesized attributes. Figure 2 shows a corresponding S-attributed grammar. The synthesis starts with the collection of $\langle annotation \rangle$ names. $\langle annotation \rangle$ elements delegate their *name* to the enclosing $\langle modifiers \rangle$ element which collects them. For $\langle classbody \rangle$ and $\langle interfacebody \rangle$, the *used* attribute is set to true if the collected set of modifiers contains the *@Transactional* annotation.

The value of the *used* attribute is then propagated to the $\langle type \rangle$ declaration. The *used* attribute value for annotation types is always false since they cannot use the *@Transactional* semantics. The $\langle typedecl \rangle$ propagates the *used* value to the enclosing $\langle typedecllist \rangle$. In addition, it is checked whether the type declaration itself uses the *@Transactional* annotation. Besides that, it is checked whether the type declaration is the actual Spring context configuration class and if so, whether the transaction management is enabled or not. The *enabled* attribute represents the value of that check.

The semantic rules of $\langle typedecllist \rangle$ collect the attributes of each enclosed type declaration. The attributes are set to true if they are true for at least one type declaration. The attributes are then validated at the root production. An error is detected, if the transaction management is enabled but no method uses the *@Transactional* annotation.

Consider again Listing 1.1 where an error occurs if the transaction management is enabled but not used because no methods are annotated with *@Trans-*

$\langle annotation \rangle$ ::= ‘@’ $\langle identifier \rangle$ $\quad \{\$0.name \leftarrow value(\$1);\}$

$\langle modifiers \rangle$ ::= ‘public’ $\langle modifiers \rangle$ $\quad \{\$0.names \leftarrow \$1.names;\}$
| ‘private’ $\langle modifiers \rangle$ $\quad \{\$0.names \leftarrow \$1.names;\}$
| $\langle annotation \rangle$ $\langle modifiers \rangle$ $\quad \{\$0.names \leftarrow union(newSet(\$1.name), \$2.names);\}$
| ε $\quad \{\$0.names \leftarrow emptySet();\}$

$\langle classbody \rangle$ ::= $\langle modifiers \rangle$ $\langle type \rangle$ $\langle identifier \rangle$ $\langle classbodytype \rangle$ $\langle classbody \rangle$
$\{\$0.used \leftarrow intersects(newSet($‘Transactional’$), \$1.names);\}$
| ε $\quad \{\$0.used \leftarrow false;\}$

$\langle interfacebody \rangle$ ::= $\langle modifiers \rangle$ $\langle type \rangle$ $\langle identifier \rangle$ $\langle methoddecl \rangle$ $\langle interfacebody \rangle$
$\{\$0.used \leftarrow intersects(newSet($‘Transactional’$), \$1.names);\}$
| ε $\quad \{\$0.used \leftarrow false;\}$

$\langle classdecl \rangle$ ::= ‘class’ $\langle identifier \rangle$ $\langle superclass \rangle$ $\langle interfaces \rangle$ ‘{’ $\langle classbody \rangle$ ‘}’
$\{\$0.used \leftarrow \$4.used;\}$

$\langle interfacedecl \rangle$ ::= ‘interface’ $\langle identifier \rangle$ $\langle superinterface \rangle$ ‘{’ $\langle interfacebody \rangle$ ‘}’
$\{\$0.used \leftarrow \$3.used;\}$

$\langle typedecltype \rangle$ ::= $\langle annotypedecl \rangle$ $\quad \{\$0.used \leftarrow false;\}$
| $\langle classdecl \rangle$ $\quad \{\$0.used \leftarrow \$1.used;\}$
| $\langle interfacedecl \rangle$ $\quad \{\$0.used \leftarrow \$1.used;\}$

$\langle typedecl \rangle$ ::= $\langle modifiers \rangle$ $\langle typedecltype \rangle$
$\{\$0.used \leftarrow \$2.used \parallel intersects(newSet($‘Transactional’$), \$1.names);$
$\$0.enabled \leftarrow intersects(newSet($‘Configuration’$), \$1.names)$
$\&\&\ intersects(newSet($‘EnableTransactionManagement’$), \$1.names);\}$

$\langle typedecllist \rangle$ ::= $\langle typedecl \rangle$ $\langle typedecllist \rangle$
$\{\$0.enabled \leftarrow \$1.enabled \parallel \$2.enabled; \quad \$0.used \leftarrow \$1.used \parallel \$2.used;\}$
| ε $\quad \{\$0.enabled \leftarrow false; \$0.used \leftarrow false;\}$

$\langle root \rangle$ ::= $\langle typedecllist \rangle$ $\quad \{if(\$1.enabled\ \&\&\ !\$1.used)\{error();\}\}$ | \$

Fig. 2. S-attributed Grammar for detection of `Transactional` error (excerpt).

actional, i.e. when line 10 is removed from the listing. An excerpt of the abstract syntax tree (AST) annotated with the results from applying the before-discussed attributed grammar to Listing 1.1 is provided in Figure 3. The rounded rectangles represent attributes and their values belonging to a node and the dotted lines illustrate the bottom-up flow of the computation. The excerpt depicts the evaluation of the *SpringConfig* class declaration where the value of the *used* attribute is false since the declaration does not contain any methods annotated with *@Transactional*. However, the value of the *enabled* attribute is true because the class declaration is a Spring configuration and the transaction management is enabled.

Other errors above the method level can be detected by similar attributed grammars. For errors in group II, we use L-attributed grammars.

Fig. 3. Annotated parse tree for SpringConfig (excerpt).

For errors that occur *below* the method level, we transform the abstract syntax tree to a *control-flow graph* (CFG) [ALSU07], which is more suitable for dealing with the dynamic behavior of the code. On the CFG, we then perform a *reaching definitions* analysis [Muc97, p.218].

Using this analysis, we can e.g. detect the error regarding destruction methods of prototype-scoped beans as presented in Listing 1.3.

4 Prototypical Implementation

Our approach is based on the attributed grammars explained above. Technically, the corresponding analysis has been implemented using the *pluggable annotation processing API*. This API is specified by Java Specification Request (JSR) 269 and enables the processing of annotations at compile-time [Dar06]. It defines a language model representing the source code of a processed Java project. The design of the language model can be described by the composite design pattern ([GHJV95, p.183]).

Besides that, the API defines how compiler extensions can be declared and executed. Figure 4 illustrates the architecture of the Java compiler. It is based on a pipe and filter architecture as described in [LL12, p.432]. The extension depicts how our prototype is plugged into the compilation process via the pluggable annotation processing API. Once the compiler finishes the lexical and syntactical analysis of the source code, it invokes the prototype through the plugin interface. The *SpringAnnotationProcessor* component uses the aforementioned composite structure provided by the compiler to perform analyses based on the attributed grammars as described in Section 3.

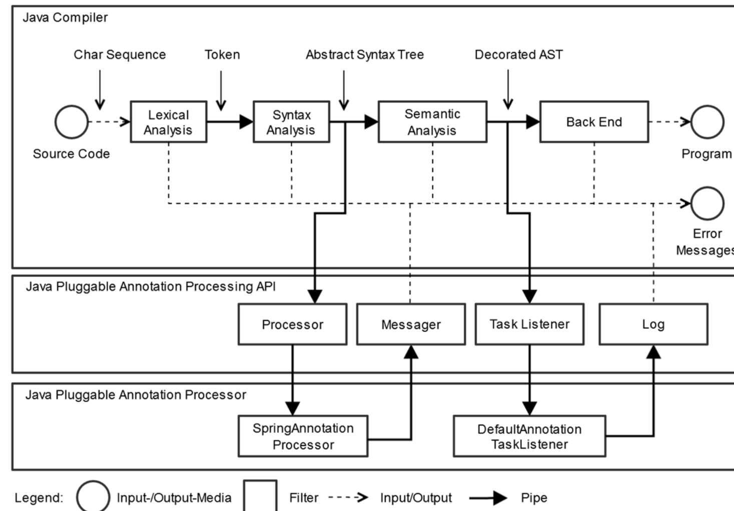

Fig. 4. Pipe and Filter Architecture of the Java Compiler

Detected errors are delegated to the *Messager* component which adds them to the compilers error messages. Next, the semantic analysis of the compiler is performed and the resulting decorated AST is passed to the prototypes *DefaultAnnotationTaskListener* which then utilizes the data-flow based analysis as described in Section 3. Again the detected errors are delegated to the compiler.

It has to be noted that the before-mentioned language model only represents a subset of Java. Language constructs which are embedded within method bodies such as assignments or method invocations are not included. Another API is required in order to access language constructs below the method level. Oracle's javac compiler provides a compiler-specific API called *compiler tree API* [Ora14]. This lower-level API provides a composite structure that represents the required *whole* AST created by javac.

5 Evaluation

The developed prototype has been used in several Java projects in order to demonstrate its functionality and to evaluate its performance.

All example applications are based on the Spring framework 3.11: The *Spring Pet Clinic*[3] is a sample application provided by the Spring framework. It is selected because it demonstrates the usage of all Spring features which are considered by the annotation processor such as annotation-based dependency injection, transaction management and caching. *Broadleaf Commerce*[4] is an open-source e-commerce framework based on Java and Spring, which consists of about 115.000

[3] See https://github.com/spring-projects/spring-petclinic

[4] See https://github.com/BroadleafCommerce/BroadleafCommerce

lines of code (LoC). It represents an actual real-world use case in which developers build modules of comparable size multiple times a day. Besides that, 12 examples[5] are considered, each of which includes an instance of one of the identified Spring-configuration error types. They do not represent practical applications but are used to ensure that the prototype is actually able to detect such errors.
Since the prototype has to be integrated into the third party projects (Spring Pet Clinic, Broadleaf Commerce), it is also possible to evaluate the integration efforts from the user perspective.

In order to assess the performance of the prototype, the build times of the projects including and excluding the prototype are compared to each other. Since all considered projects are based on Maven, the time measured by Maven itself is used. To ensure the comparability of the measured times, all builds are performed on the same machine and unchanged configuration. In addition, multiple builds are performed to minimize the possible influence of external factors of other processes performed by the operating systems.
Each project is build 41 times whereby the first build fulfills two functions: First, it is used by Maven to download and manage third party dependencies, which potentially falsifies the results. Second, the Java virtual machine requires some time for initialization when started, which also potentially falsifies the results. 40 additional builds are used to actually measure the build times. 20 of these builds are performed with the prototype and 20 are performed without the prototype.

The conducted tests reveal the following results. No Spring-related errors have been deteted within the Spring Pet Clinic and Broadleaf Commerce projects, which is not surprising, since they are sufficiently mature. In contrast, the errors placed on purpose in the example projects are detected.

The runtimes required to build the projects with and without the annotation processor differ by an acceptable amount (see Table 3). The time differences are less than one second for the small projects. Tests with an empty annotation processor, which performs no checks, show similar results. Hence, it is likely that a large part of the difference results from locating and initializing the annotation processor. The largest absolute difference observed at the Broadleaf Commerce project is ∼2 seconds, a 3% increase, for $115,000$ LoC. However, it has to be noted that the project consists of seven modular projects and the Java plugin is invoked individually for each of them. Hence, the annotation processor is seven times located and initialized.

Up to now, our prototype handles 12 of the identified error types and, as our experiments have shown, it is able to detect instances of these error types successfully. The implementation of the detection of remaining error types is still pending.

In order to improve the acceptance of our tool, the prototype is required to minimize the occurrences of false reports. The following can be stated w.r.t. the correctness and completeness of our tool. All errors above the method level are reported properly and there are no reports of errors which actually don't occur.

[5] See https://github.com/vvhof/DetectingSpringConfigurationErrorsExamples

		Avg. Build Times in ms		
Sample Project	**LoC**	**Disabled**	**Enabled**	**Diff.**
Error Type 1	29	2 568	2 971	16%
Error Type 2	156	2 675	2 932	10%
Error Type 3	36	2 544	2 741	8%
Error Type 4	37	2 535	2 825	11%
Error Type 5	37	2 524	2 870	14%
Error Type 6	69	2 524	2 915	15%
Error Type 7	56	2 558	2 756	8%
Error Type 8	53	2 501	2 629	5%
Error Type 9	39	2 463	2 725	11%
Error Type 10	39	2 469	2 668	8%
Error Type 11	54	2 502	2 764	11%
Error Type 12	54	2 516	2 748	9%
Spring PetClinic	1 390	9 912	11 448	15%
Broadleaf Commerce	115 902	55 108	56 970	3%

Table 3. Build Times with enabled and disabled prototype.

Below the method level, we are using a static analysis based on the control-flow graph in addition to an attributed grammar. Due to the unavoidable loss of precision in that analysis, it may happen that errors are reported, which cannot occur thanks to data dependencies which the reaching definitions analysis ignores. Fortunately, such errors rarely happen in practice, since it is bad programming style to let the correctness of the configuration depend on the control and data flow. One may even argue that such cases should be reported. Our current implementation does not yet support inter-method analysis. Thus, errors which can only be detected with such an analysis are currently not yet covered.

There are two limitations of our approach. As explained above, the limits of static code analysis are also the limits of our approach. Second, the pluggable annotation processing API restricts the usage of annotation processors to a Java compiler. The usage of the compiler tree API also binds the annotation processor to Oracle's specific Java compiler javac.

6 Conclusion and Future Work

Dependency injection is an elegant design pattern. However using it, configuration errors may occur which available Java compilers cannot detect. These erroneous configurations are hence only detected at runtime, which requires difficult debugging and causes nasty delays during the development of software. We have developed a compiler-plugin for the javac compiler which is able to find such configuration errors at compile time. Conceptually, the plugin is based on attributed grammars and the Java pluggable annotation processing API.

For the popular framework Spring and based on a literature review and expert interviews, we have first of all collected a set of 38 possible types of configuration

errors. Then, we have classified these error types into four categories. The classification depends on two aspects. First, we check whether an error requires an analysis above or below the method level. Secondly, we check whether the error is depending on the Spring context or not. For each of these classes of errors, we have developed a scheme for an S- or L-attributed grammar and instantiated this scheme for every considered possible error. By combining the attributed grammars of each error type, we obtain one large L-attributed grammar. For errors which require an analysis of the control flow, a reaching definitions analysis based on the control-flow graph has been added.

In experiments based on a couple of small and two big applications, we have evaluated that the compiler plugin produces an acceptable runtime overhead. Moreover, it was able to find all configuration errors which we have inserted. Due to the imprecision of the reaching definitions analysis, false positives may happen in principle. In practice, this did not happen.

Our plugin is a valuable tool for Spring developers and used in practice by our project partner in industry. It has helped to speedup software development using Spring considerably.

Our current implementation handles 12 out of 38 identified types of errors. As future work, we would like to extend the plugin such that the remaining error types are also covered. For all but 5 error types, this will be straightforward and we just have to instantiate our general schemes again. The remaining 5 errors will require some inter-method analysis.

7 Acknowledgments

We thank our project partner viadee GmbH for the fruitful collaboration.

References

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools.* Addison-Wesley Publishing Company, USA, 2nd edition, 2007.

[APM+07] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[Bur03] Oliver Burn. Checkstyle, 2003.

[CK05] David R Cok and Joseph R Kiniry. Esc/java2: Uniting esc/java and jml. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer, 2005.

[Dar06] Joseph D. Darcy, 2006.

[EH07] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[GF07] David Greenfieldboyce and Jeffrey S Foster. Type qualifier inference for java. In *ACM SIGPLAN Notices*, volume 42, pages 321–336. ACM, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Joh15] Rod et al. Johnson. Spring framework reference documentation, 2015.

[Knu68] Donald E. Knuth. Semantics of context-free languages. In *Mathematical Systems Theory*, pages 127–145, 1968.

[LL12] Jochen Ludewig and Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2012.

[MME+10] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. Javacop: Declarative pluggable types for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):4, 2010.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Ora14] Oracle Corporation. Openjdk compiler tree api specification, 2014.

[Ora15] Oracle Corporation. "package java.lang.reflect", 2015.

[PAC+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[Piv15] Pivotal Sofware, Inc. Spring framework, 2015.

[PMD15] PMD. Pmd, 2015.

[Pra09] Dhanji R. Prasanna. *Dependency Injection*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[SDE12] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[Son15] SonarSource S.A. Sonarqube, 2015.

[VWKBJ06] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and Eric Johnson. Adding domain-specific and general purpose language features to java with the java language extender. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 728–729, New York, NY, USA, 2006. ACM.

[Wal14] Craig Walls. *Spring in Action*. Manning Publications Co., Greenwich, CT, USA, 4th edition, 2014.