IntelliJML: A JML Plugin for IntelliJ IDEA

Steven Monteiro
s.c.monteiro@student.utwente.nl
University of Twente
Enschede, The Netherlands

Erikas Sokolovas
e.sokolovas@student.utwente.nl
University of Twente
Enschede, The Netherlands

Ellen Wittingen
e.m.wittingen@student.utwente.nl
University of Twente
Enschede, The Netherlands

Tom van Dijk
t.vandijk@utwente.nl
University of Twente
Enschede, The Netherlands

Marieke Huisman
m.huisman@utwente.nl
University of Twente
Enschede, The Netherlands

ABSTRACT

Java code can be annotated with formal specifications using the Java Modelling Language (JML). Previous work has provided IDE plugins intended to help write JML, but mostly for the Eclipse IDE. We introduce IntelliJML, a JML plugin for IntelliJ IDEA, with a focus on ease of use and maintainability. Features such as syntax, semantic, and type checking, as well as syntax highlighting and code completion are integrated into the plugin. The plugin can also be extended in the future to add more features. The source code for the plugin can be found at https://gitlab.utwente.nl/fmt/intellijml.

CCS CONCEPTS

• **Software and its engineering** → *Software verification.*

KEYWORDS

JML, Java Modelling Language, IntelliJ IDEA, Error checking, Plugin, Java, IDE, Interactive Development Environment

ACM Reference Format:
Steven Monteiro, Erikas Sokolovas, Ellen Wittingen, Tom van Dijk, and Marieke Huisman. 2021. IntelliJML: A JML Plugin for IntelliJ IDEA. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '21), July 13, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3464971.3468423

1 INTRODUCTION

The Java Modelling Language (JML) [10] is a specification language that is used to specify the intended behaviour of Java programs. With IntelliJ IDEA [8] recently enjoying high popularity [14, 16], we identified the need for a plugin for JML that supports IntelliJ. Previous JML plugins do not provide support for IntelliJ [4, 15] or only provide a basic wrapper around existing JML tools [6].

To address this need we develop a new JML plugin for IntelliJ called IntelliJML (Figure 1). It is primarily made as a teaching tool in the Software Systems course at the University of Twente. However, the plugin could prove a useful tool for anyone working with JML.

Table 1: Supported features for each plugin

	IntelliJML	OpenJML	KeY
Syntax, semantic & type checking	✓[1]	✓	✓
Runtime checking[2]	×	✓	✓
Static checking[3]	×	✓	✓
Code completion	✓	~[4]	✓
Syntax highlighting	✓	×	✓
Java versions	8 and up [5]	8	8 and up [6]
IDE support	IntelliJ IDEA	Eclipse	Eclipse

The requirements for the plugin are: IntelliJ support, forward compatibility with future Java versions, extensive error checking, maintainability and user friendliness. Based on these requirements, we focus on the user-facing side of JML such as descriptive error messages and code completion. Most language level 0 features [10] (page 18-21), the keyword "pure" and quantified expressions are supported by the plugin. All Java versions starting from version 8 are supported. Our plugin does not provide static or runtime checking. The requirement of forward Java compatibility proved to be a unique challenge - necessitating the use of combination lexers and a manually written parser extension. Because there was limited time to write the plugin, it is written in such a way that it can be extended in the future with more features.

2 RELATED WORK

There already exist several plugins for JML. The most relevant are OpenJML [3, 4], KeY [1, 15] and an IntelliJ plugin called OpenJML/ESC [6]. OpenJML and KeY are both Eclipse [5] plugins. OpenJML only supports Java 8, whereas KeY also supports Java versions higher than 8. However, the authors of KeY indicate that it does not support all features of Java, such as generics and lambdas. So these plugins differ from our plugin in what IDEs and Java versions are supported. OpenJML/ESC does support IntelliJ. However, it has not been updated since 2018 and our analysis of the source code

[1] Supports most language level 0 features, the keyword "pure" and quantified expressions.
[2] By means of calling external command-line tools.
[3] See footnote 2
[4] Limited keyword completion.
[5] Lambda's are allowed but not type checked.
[6] Does not support all features of Java, such as generics and lambdas.

FTfJP '21, July 13, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8543-5/21/07…$15.00
https://doi.org/10.1145/3464971.3468423

```
// returns a list with boxes that have a volume >= the minimum volume specified
/*@ ensures (\forall int i; 0 <= i && i < boxes.size();
        boxes.get(i).getVolume() >= minimumVolume ==> \result.contains(boxes.get(i)));
  */
/*@ ensures \result.size() <= boxes.size();
  @ requires minimumVolume > 0;
  */
//@ pure;
public List<Box> getAllBoxesWithMinVolume(float minimumVolume) {
    List<Box> goodBoxes = new ArrayList<>();
    for (Box box : boxes) {
        if (box.getVolume() >= minimumVolume) goodBoxes.add(box);
    }
    return goodBoxes;
}
```

Requires clauses must be placed before all other clauses in a specification as it is a pre-condition

Figure 1: A screenshot of the plugin displaying an error in IntelliJ IDEA.

indicates that it is a wrapper around OpenJML and does not offer syntax highlighting nor code completion like our plugin does.

A short overview of the features of the plugins is given in Table 1. It should be noted that the KeY and OpenJML plugins use external command-line tools for runtime and static checking [1, 3, 4, 15].

3 INTELLIJ PLATFORM FEATURES

The IntelliJ platform provides an extensive language development API. IntelliJ has decoupled language specific logic into plugins [2] and the platform itself provides a generic abstract syntax tree (AST) implementation for plugins to work with. This interface is known as PSI (Program Structure Interface) and provides tools for implementing syntactic and semantic language features [9]. The platform also provides features for the construction of the ASTs by allowing the use of either ANTLR [13] grammars or IntelliJ's own grammars using the Grammar-Kit plugin [7].

4 ARCHITECTURE

We illustrate the basic flow of a JML comment through the plugin in Figure 2.

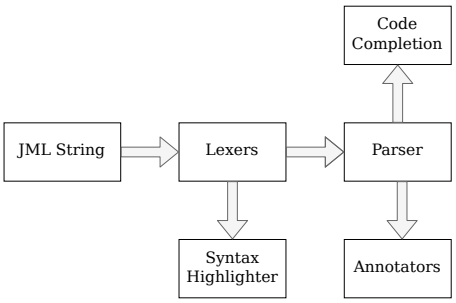

Figure 2: The flow of data through the plugin.

When receiving a JML string, it is first passed through several lexers. These lexers are layered together and split the string into JML and Java tokens. Layering of multiple lexers was necessary as IntelliJ's native Java lexer is invoked for lexing Java expressions inside JML statements. IntelliJ's lexer is used to ensure future Java version compatibility. In the first layer all JML keywords are lexed as individual JML tokens. Then in the second layer tokens belonging to a Java expression are merged into a single token that represents that expression. Afterwards, the next layer splits the Java expression token into separate Java tokens. Finally, the last lexer replaces and merges certain Java tokens, this is done to account for JML keywords that can be mixed with Java expressions. This process is illustrated in figure 3.

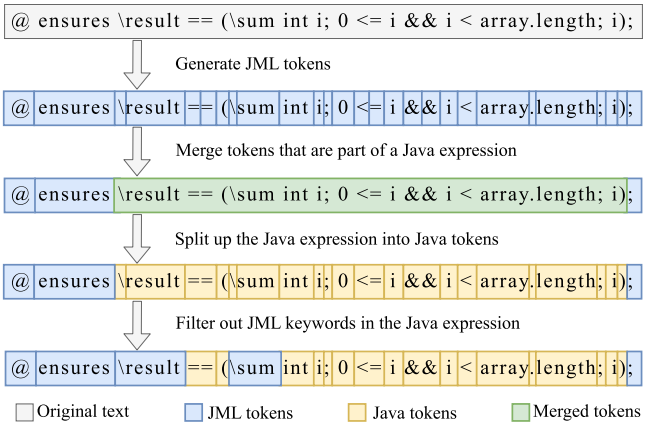

Figure 3: The flow of a JML comment through the lexers.

The output of these lexers is used by the syntax highlighter and the parser. The syntax highlighter applies colours to all generated JML tokens. IntelliJ's native highlighting colours are used for generated Java tokens.

The parser takes the generated tokens and generates a parse tree. This parser was generated from a custom grammar in Backus-Naur form (BNF) [11] describing the relevant subset of the JML syntax. For Java expressions embedded in JML we have written a parser extension that recursively generates tree elements for JML inside

the embedded Java expressions and is called by the generated parser in places where a Java expression is expected.

The generated tree is used by annotators to perform error checking. If an error is encountered, a message will be displayed to the user. There are several types of messages: syntax checking, semantic checking, and type checking. Type checking also checks whether references to fields, methods, classes, etc. can be resolved.

To check whether references can be resolved, IntelliJ's Java API is used, which returns the closest match to a reference. It is checked that this match is valid: It should, for example, have the correct visibility. This visibility checking consists of querying the Java modifiers of the match such as "private" and "public". When visibility is incompatible, JML comments belonging to the match are checked, to see whether they contain "spec_public" or "spec_protected", as those change the visibility for JML specifications.

To perform type and semantic checking on JML expressions embedded in Java such as "\result", "\old()", "\typeof()", etc. replacements are done to convert them to Java code. This Java code is then checked using IntelliJ's Java API. For example, "\old(expression)" is simply replaced with "expression". This is done recursively since the expression can still contain JML that needs to be replaced. "\result" is replaced with the default value for primitive types and replaced with "((RefType) null)" for reference types.

The plugin also performs checks for common mistakes. For example, when a user puts whitespace between the start of the comment and the first @ character a warning is given, telling the user that the comment is not valid JML.

The annotators have a hierarchy and run in sequence. If one annotator finds that an error needs to be displayed for a certain token, it will stop further annotation for that token. This avoids confusion for the user as this prevents multiple messages being displayed for the same token.

Code completion allows for the completion of both JML and Java expressions for most things a user could write, such as: keywords, common code boiler plate (e.g. quantified expressions), chained method calls, etc. All supported JML expressions have proper completion, but not all Java expressions e.g. lambda expressions. This is because IntelliJ's own Java completion service could not be reused and a custom one had to be written.

5 MAINTAINABILITY

As mentioned in Section 1, maintainability is an important requirement for our plugin. The Java programming language is under constant development [12]. New language features may therefore be added. We take special care to use the IntelliJ public API to interact with Java code rather than a custom solution wherever possible. This aims to ensure that minimal maintenance is required to the plugin's code, relying on the support for future versions being provided by IntelliJ instead. Being a commercial product, we expect IntelliJ's support to be adequate.

The plugin has extensive unit tests that achieve 82% class coverage, 72% method coverage and 80% line coverage. This increases maintainability as incompatible behaviour changes made to the JetBrains API or plugin source code should be caught by unit tests.

We also provide a maintainers' guide that details the inner workings of the plugin. The source code of the plugin has extensive Javadoc that explains the functionality of each section of the code. This extensive documentation should allow developers to more easily and quickly become familiar with the internals of the plugin, improving maintainability.

Code decoupling patterns are applied where appropriate to make changing the source code of the plugin simple, which further contributes to maintainability.

6 FUTURE WORK

We believe it to be relatively straightforward to extend the supported JML subset, as it would only involve additions to the internal grammar and the error checkers. For some of the more complex JML features one might also need to extend our parser extension and lexers.

A framework was set up for runtime assertion checking which can transparently insert custom assertions into Java code. It currently can not generate assertions, but with the current infrastructure the remaining task is translating JML to valid Java assertions.

A possible feature addition could also be static checking. The easiest way to add this support would be to write a wrapper around an existing static checker. Such static checkers can be found in, for example, OpenJML and KeY.

Code completion could also be expanded on as not all Java syntax is properly supported, such as lambda expressions or local method overrides. Additionally, some convenience features could be implemented such as auto-import or expanded code templates.

7 CONCLUSION

We created a new JML plugin for IntelliJ that focuses on maintainability and forward Java compatibility. The plugin supports the core parts of the JML syntax, which can be expanded upon relatively easily with the existing code infrastructure. It is currently a JetBrains IDE front-end for JML providing no specification checking capabilities. Specification checking could be introduced into the tool by writing a wrapper for an existing command line checker, e.g. the KeY project's command line static verifier tool.

ACKNOWLEDGMENTS

We thank the student assistants who provided useful feedback that improved the quality of our work.

REFERENCES

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, and M. Ulbrich (Eds.). 2020. *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*. Lecture Notes in Computer Science, Vol. 12345. Springer. https://doi.org/10.1007/978-3-030-64354-6

[2] N. Chashikov. 2018. Extract Java support to a separate plugin in IntelliJ IDEA. https://youtrack.jetbrains.com/issue/IDEA-195719

[3] D. Cok. 2014. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014 (EPTCS, Vol. 149)*, C. Dubois, D. Giannakopoulou, and D. Méry (Eds.). 79–92. https://doi.org/10.4204/EPTCS.149.8

[4] D. Cok. 2018. OpenJML. https://www.openjml.org/

[5] Eclipse Foundation. 2021. *Eclipse*. Eclipse Foundation. https://www.eclipse.org/

[6] S. Gonschorowski. 2018. OpenJML/ESC. https://plugins.jetbrains.com/plugin/11072-openjml-esc

[7] JetBrains s.r.o. 2021. *Implementing a Parser and PSI: IntelliJ Platform Plugin SDK*. JetBrains s.r.o. https://plugins.jetbrains.com/docs/intellij/implementing-parser-and-psi.html

[8] JetBrains s.r.o. 2021. *IntelliJ IDEA*. JetBrains s.r.o. https://www.jetbrains.com/idea/

[9] JetBrains s.r.o. 2021. *Program Structure Interface (PSI): IntelliJ Platform Plugin SDK*. JetBrains s.r.o. https://plugins.jetbrains.com/docs/intellij/psi.html

[10] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. 2013. *JML Reference Manual.* https://www.cs.ucf.edu/~leavens/JML/refman/jmlrefman.pdf
[11] D. D. McCracken and E. D. Reilly. 2003. *Backus-Naur Form (BNF).* John Wiley and Sons Ltd., GBR, 129–131.
[12] Oracle Corporation. 2021. Oracle Java SE Support Roadmap. https://www.oracle.com/java/technologies/java-se-support-roadmap.html
[13] T. Parr. 2013. *The Definitive ANTLR 4 Reference.* The Pragmatic Programmers, LLC.
[14] Stack Overflow. 2019. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019
[15] The KeY Project. 2018. Program Verification. https://www.key-project.org/applications/program-verification/ Accessed at 23th April 2021.
[16] B. Vermeer. 2021. IntelliJ IDEA dominates the IDE market with 62% adoption among JVM developers. https://snyk.io/blog/intellij-idea-dominates-the-ide-market-with-62-adoption-among-jvm-developers/