Guide for the maintainer

This document is provided as a guide to anyone who needs to work with the source code of the IntelliJML plugin. It is intended to be a technical overview of the plugin's inner working, not an explanation of design choices. It is written in such a way that it is not necessary to read the project's full Design Report to understand this guide.

The reader is warned that the IntelliJ API is highly undocumented and therefore not pleasant to work with. The IntelliJ bug tracker states "Why haven't we done [Javadoc] yet: Because a) it creates the illusion that we have documentation" [1]. The limited documentation that is available can be found at [2].

1 Tools

Working with the IntelliJML source code requires the following tools:

- Java 8 or higher (set to language level 8 when higher).
- IntelliJ IDEA 2020.1 or higher. Attempting to work with the source code in any other IDE is highly discouraged and not covered by this document.
- The Gradle plugin for IntelliJ.
- The Grammar-Kit plugin for IntelliJ.
- Optional but highly recommended: The PsiViewer plugin for IntelliJ.

2 Version support

IntelliJML is written to be source-compatible with Java 8 and IntelliJ IDEA 2020.1. Unless there is a good reason to increment the minimum IDE version, it is recommended to keep it this way. The version of Java that the plugin is compiled with does not affect the plugin's support for language features, since these are processed through IntelliJ's API. Note that IntelliJ versions older than 2020.3 can be run with Java 8 [3], so incrementing the project's Java version also implies dropping support for these IDE versions. Support for 2019.3 and older is possible, but requires reworking the use of API methods that were introduced in newer versions, which is unlikely to be worth the effort.

IntelliJ version support can be configured by setting `pluginSinceBuild` and `pluginUntilBuild` in `gradle.properties`. If the IDE version is not within this range, IntelliJ will refuse to load the plugin. Before changing these values, support should be verified using the Gradle task `runPluginVerifier` . This task checks for the use of deprecated and nonexistent classes and methods for all IDE versions listed at `pluginVerifierIdeVersions` in `gradle .properties`. Its output can be viewed at `build/reports/pluginVerifier`.

To package the plugin for distribution, simply run the `jar` Gradle task and copy the output from `build/libs`.

3 Data flow

In section **??**, the data flow going from a string containing JML to each of the plugin's features was explained on a surface level. In Figure 1, the same data flow is shown more in-depth.

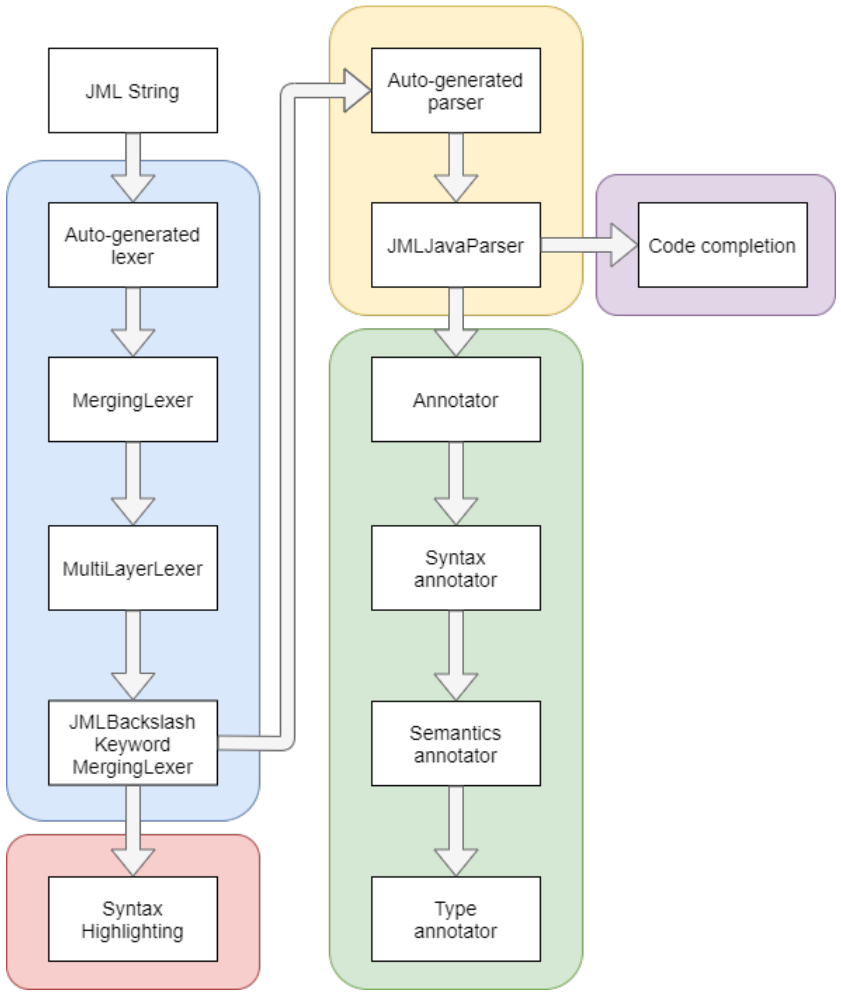

Figure 1: The flow of data through the plugin

4 Language embedding

4.1 Embedding JML into Java

The topmost level of the language embedding ensures that JML in Java comments is recognized as such by IntelliJ. To do this, `JMLMultiHostInjector` listens for comments in Java files. If one is found, and the first non-whitespace character after the start of the comment is an at-sign, the comment will be marked as JML, which causes IntelliJ to trigger the part of the plugin that processes JML strings.

This form of injection through the IntelliJ API automatically adds a green background to the embedded part, which in the case of this plugin means that all JML specifications have a green background. This is obviously undesirable, but IntelliJ does not provide an option to disable this. To fix this, the green background is manually overridden with one that has the same color as the normal background, effectively making it invisible.

4.2 Embedding Java into JML

Java was embedded back into JML (because JML is already being embedded into Java) by means of an extension of the generated JML parser and a multilayered lexer. This complicated approach was necessary due to limitations of the IntelliJ API that explicitly forbids embedding another language into an already embedded language. This essentially meant that a ground-up approach was needed.

First, the generated JML lexer is adapted to work with the IntelliJ API by means of a `FlexAdapter` class that envelopes the generated lexer (the generator names the class `_JMLLexer`) and adapts the interface to work with the API. This all happens in the `JMLMergingLexer` class that also contains the logic for collecting all the individual tokens that constitute a single mixed JML-Java section of JML and returning that entire section as a single token. This was needed because the generated JML lexer does not contain any Java tokenization, basically it returns nonsense in this section. To fix this, the single token is passed to the `JMLMultiLayerLexer`. This lexer dynamically alternates between the `JMLMergingLexer` for lexing the pure JML parts and `JavaLexer` to handle the Java parts (all of which is in the large combined token). This way, the single large token is broken up into a stream of mixed Java and JML tokens. The output of this lexer is then fed into another merging lexer, the `JMLInJavaExprMergingLexer`. This lexer merges/replaces tokens that were previously interpreted as Java into their appropriate JML tokens (JML implication operators and JML keywords with backlashes). Using this method, there is now a correct stream of JML-Java tokens.

Next, this token stream is passed to the JML parser. The generated parser handles the pure JML sections as specified in the BNF grammar, but the mixed JML-Java sections are handled by the custom extension to it. The custom parser extension is the `JMLJavaParser` class (technically this is called a parser util class, but it acts as a parser extension in our instance).

This class takes care of building the actual parse tree for the mixed parts. The parser extension is called by the generated parser when it reaches the `java_expr` node in the generated parser. In the BNF it can be seen that the `<<parseJavaExpression>>` function call is given instead of a regular rule for the `java_expr` node.

This is not an ideal solution, as the parser extension does not provide a proper parse tree for the Java expressions and just dumps the Java tokens under the appropriate JML node. This is, again, because forward compatibility with future Java versions is needed, so implementing custom parsing logic is not possible.

This unfortunately means that both the annotator and code completion services rely on string evaluation of Java expressions quite heavily in certain places, which is inefficient. If a proper Java parse tree could be received, this issue might be fixable.

It might be possible to retrieve a proper Java parse tree for the Java tokens. In the extension section of the parser, it is possible to dynamically switch to the IntelliJ Java parser and dynamically switch back to the calling parser. This was unsuccessfully briefly attempted, but the project was already behind schedule by several weeks because of this embedding issue. More time could not be put into investigating this possible solution, as the current solution allowed the project to move forward.

4.3 Lexing and parsing

4.3.1 Re-generating the lexer and parser

For the lexing and parsing, a grammar was written in BNF form, which can be found in `main/grammar/JML.bnf`. This grammar is used to auto-generate the lexer and the parser. To generate the lexer, right-click the `JML.bnf` file (the grammar-kit plugin in IntelliJ is needed for this), and click `Generate JFlex Lexer`. A window should open that allows choosing where to store it. Store it in `main/gen/nl/utwente/jmlplugin/parser` under the name `_JMLLexer.flex` . Then right-click that file, and click on `Run JFlex Generator`. Then, go back to the `JML.bnf` file, right-click it, and now select `Generate Parser code`. After that, navigate to the `psiToOverwrite` and copy the two files in there, and paste them in `main/gen/nl/utwente/jmlplugin/psi`.

4.4 Structure of the lexer

The lexer consists of 4 layers. The first layer is the auto-generated lexer, which converts raw strings to JML tokens. The second layer, the class `MergingLexer`, merges tokens that are part of a Java expression inside JML into one token. After that, `JMLMultiLayerLexer` takes the merged Java expression token and separates it into Java tokens. However, the Java expressions can contain JML keywords such as `\result`, which are not valid Java. For that reason, there is a fourth layer that turns those back into JML tokens (`JMLInJavaExprMergingLexer`). A quick demonstration of the lexers can be seen in Figure 2. The blue sections are JML tokens, the green section is the big Java expression token and the yellow sections are Java tokens.

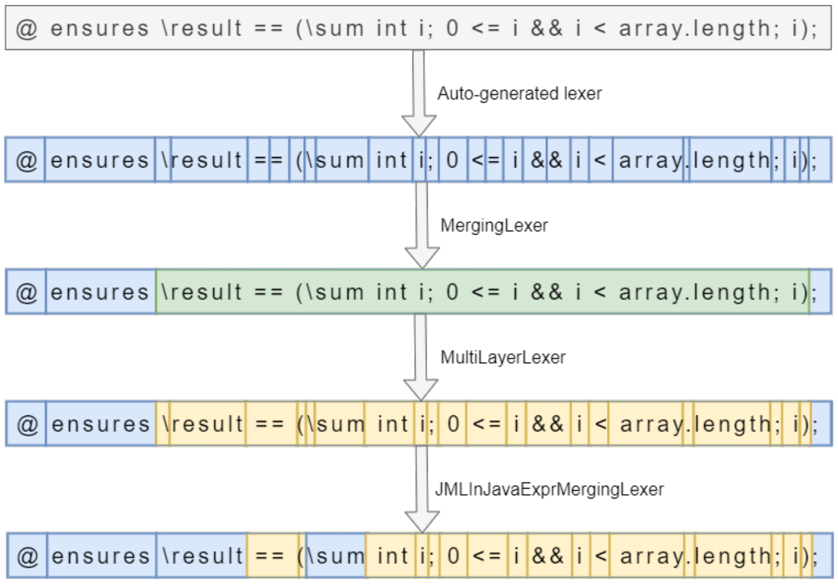
@ ensures \result == (\sum int i; 0 <= i && i < array.length; i);
Auto-generated lexer
MergingLexer
MultiLayerLexer
JMLInJavaExprMergingLexer

Figure 2: The flow of a JML comment through the lexers

4.5 Structure of the parser

There are two parsers. The first parser that is run is the auto-generated one. This parser does all the JML parsing, except for the Java expressions inside JML. For those, the auto-generated parser calls the manual parser, which can be found in `main/java/nl/utwente/jmlplugin/parser/JMLJavaParser`. This parser parses the Java tokens and JML expressions inside the Java expression, and creates a tree for them.

5 Annotation

Slightly misleadingly named, annotations refer to IntelliJ's warning and error messages, regardless of the nature of those messages. IntelliJML contains a multi-stage annotator (see `nl.utwente.jmlplugin.annotator`) that checks and provides warnings and errors for syntax, semantics, and typing, in that order. it is important to note is that the parser also provides error annotations, which are intercepted by the custom syntax annotator and adapted to be less confusing and more legible.

The annotator that is first called is the `JMLAnnotator`. In there, the `annotate` method is called for each element in the JML tree. It is checked that comment is an actual JML comment, and if it is, the syntax checker in `JMLSyntaxAnnotator` is called. If no syntax errors have been found, the semantic checker is called (`JMLSemanticsAnnotator`). If no semantic errors are found either, the type checker is called (`JMLTypeAnnotator`). Because this procedure is followed for each tree element, there can be both semantic and type errors in the same JML comment, but only if they are not on the same element in the tree. However, if the JML comment contains a syntax error, no tree elements at all are checked for semantic and type errors.

The type checker makes use of a few classes, of which the most important are mentioned here. First of all, a class called `ExtraVariables` holds a list of variables that should be accessible by the Java expressions inside JML, but cannot be resolved by the Java expression resolver. For example, the variables declared in JML quantified expressions, parameters of methods, initializers of for-loops, etc. are put in `ExtraVariables`. Another class worth mentioning is `RangeManager`. To understand why it is needed, a little more explanation is needed about the `JMLTypeAnnotator`. JML keywords can occur inside Java expressions. Because IntelliJ should create an actual expression from the JML Java expressions, those must be removed, as they are not valid Java. To achieve that, replacements are done on them. However, the replacements can be longer or shorter in amount of characters than the originals, causing the text ranges of errors to be incorrect. This means that errors are shown to the user at the wrong position. The class `RangeManager` is used to keep track of these changes, and to calculate the original position to be used in error messages.

6 Code completion

Code completion is provided both for JML keywords and inside Java expressions. The JML keyword implementation is relatively straightforward. For Java expressions, however, it is impossible to obtain a proper parse tree, which is required to be able to use IntelliJ's own Java code completion. This means that the Java code completion inside JML is mostly custom and therefore has subtle differences and possible bugs compared to regular Java code completion.

Some parts of IntelliJ completion were sufficiently generic to be be reused. The class finder, responsible for finding all visible classes in the project, is reused from IntelliJ completion, as well as the sorting algorithm that IntelliJ uses for its suggestions.

The main class responsible for completions is the `JMLCompletionContributor` class that houses the `PsiTree` patterns on which completions are called. It also registers all completion providers that provide completion suggestions when a completion is called on the corresponding `PsiTree` pattern.

The `completionProvider` classes are responsible for generating completion suggestions. Each contributor is responsible for providing different suggestions to the result set that stores all collected suggestions from all invoked completion providers. It should be noted that multiple completion providers can be invoked simultaneously, and they contribute to the result set asynchronously. For that reason, it is not possible to have synchronisation between them without risking a deadlock in the completion threads.

Most of the functionality of the completion providers was extracted and put into the `JMLCompletionUtils` class, as a lot of code is shared between the different completion providers. This means that most of the functionality is located there.

Most of the `completionProvider` classes are straightforward and reading the Javadoc should provide a good explanation of what they are doing. However, the `JMLJavaDotCompletionProvider` class is a bit of a hack that handles dot expression completion suggestions, for example `(Integer.getInteger("5"))`. The reason for this is that there was not enough time left in the project to write a recursive dot expression evaluator, as accounting for all the import and visibility nuances is complicated. Instead, the complete dot expression string is passed to the Java parser to evaluate the return type up to that point and then provide suggestions based on that. The evaluator does support support some JML-Java expression mixing, but only `\result`.

and `\old().` keywords are properly supported.

The support for these keywords was achieved by replacing each of these keywords with equivalent valid Java expressions. `\result.` is replaced by `((return_type)0)` or `((return_type)null)` in the string that is passed to the evaluator. 0 is used for primitive types (as any primitive literal is castable to any primitive) and null for object return types (as null is castable to any object). For `\old().` Just the `\old` part is removed as the parentheses are valid Java and do not need to be removed.

For accessibility and visibility checking there are two methods that should be kept in mind. First is the `JMLCompletionUtils.isAccessibleHere()` method. This method checks field, method, class and package accessibility at the given location in the project. It accounts for all Java rules regarding to accessibility. However, local variable scoping is not resolved by it, if that is needed - look at `JMLCompletionUtils.localVariableInScope()`. This is written for local variables only, but in theory it is just a much more expensive string evaluation method that also should account for the prior rules as well, it is not used everywhere as it is much more expensive than the `isAccessibleHere()` method.

7 Runtime checking

The implementation of runtime checking in the source code is incomplete. It is meant to make use of the following procedure: When a compilation is started, all source files in the project are collected and copied into a temporary directory (the one provided by the operating system). The tree of each copied Java source file is then walked to collect all JML specifications. Each JML specification is converted into an appropriate check in Java code and inserted into the tree of the corresponding copied file. Once all JML specifications have been processed, the copied files are included in the compilation scope, and the originals are excluded. The compilation is then performed by IntelliJ without intervention. After compilation, the temporary directory is cleaned up to ensure that IntelliJ cannot unintentionally show the temporary files to the user through the GUI.

In the current code base, the capability to scan all Java files and transparently insert code is functional. The missing part is the code that converts a given JML specification into an appropriate check. Currently, all code paths related to runtime checking that are present in the plugin are intentionally inaccessible because they are dependent on a setting that cannot be toggled from the GUI. To make the feature accessible, add a checkbox that enables runtime checking in `nl.utwente.jmlplugin.settings.ConfigurationPanel`. It should be noted that when enabled, the code in its current state does affect the compilation process, but does not do anything useful.

Copying the source files is a workaround best described as a hack. There is however no simple alternative for this: While IntelliJ does support programmatically inserting code at compile time, it forces these changes to be visible to the user, which would be unwanted behavior for this plugin.

If the missing parts of runtime checking are implemented, it is suggested to use if-statements that throw AssertionErrors rather than assert statements. The latter do not have any effect unless explicitly enabled by a VM argument, which has the potential to confuse students.

8 Testing

Unit tests are focused on the parser and annotators, but there are also completion tests. The test framework used is the one used internally by IntelliJ, which is itself based on JUnit 3. Integration tests are also available for the annotators. Unit tests and integration tests can be executed from the IDE as expected, and they can be found in `test/java/nl/utwente/jmlplugin`. Documents describing manual system tests are available in the project's Design Report.

Setup To run the unit and integration tests, the IntelliJ community edition repository needs to be cloned from https://github.com/JetBrains/intellij-community. This repository is multiple gigabytes in size, but is unfortunately required: Otherwise, the test runner cannot resolve references to the Java standard library in the test files due to an internal dependency. Make sure that the repository is on the master branch once it has been cloned. After that, navigate to the file `build.gradle.kts` (outside the `src` folder) and look for the `tasks { test {} }` section. Once found, replace `filepath` in `systemProperty("idea.home.path", "filepath")` with the file path of the cloned repository.

Running the tests Now tests can be run. To run all tests after each other open the Gradle tab and in `Tasks` run `verification > test`. Tests can also be run for a single component. For those, run configurations are available in the IDE. The names of these configurations start with "JML" and end with "Test" and are self-explanatory. For example, the configuration called `JMLSyntaxAnnotatorTest` tests the syntax annotator.

When all tests fail A note should be made that in rare occasions all the tests fail. This is an issue with either Gradle or IntelliJ and not with the plugin. If this occurs, first try to re-run all the tests. If that does not work, look for a folder called `build` and delete it. Then try to run the tests again. If that also does not work, one can try to also delete the `.gradle` folder in the project and in one's user folder, but be warned that it takes Gradle several minutes to rebuild in that case.

Running the tests with coverage The tests can also be run with coverage. Unfortunately, the Gradle task that runs all tests does not run properly with coverage, as it starts a new instance for every test class it runs. This has been done intentionally because there were problems with tests classes not being closed properly, which caused the tests to give different results based on the order they were run. Running them as separate instances fixed this issue. However, this causes only the coverage of the last instance to be shown, and not of all tests together. To solve this, there is a run configuration called `CoverageAllTests`, which runs all test classes after each other. To run with coverage, select `CoverageAllTests` and click `Run with coverage`. During the execution, a popup might appear asking whether coverage data should be displayed for `TestName` results. When this comes up, click on `Add to active suites`. When the task is done running, the coverage should appear on the right side of the screen. It can be verified that the coverage of all classes is being displayed by navigating to `Run > Show Coverage Data...` and then checking all the checkboxes and clicking `Show selected`.

9 Known bugs

- An at-sign can be placed after a JML clause without causing a syntax error; it should only be allowed before a clause.
- When an import in a class is only used in JML and not in the Java code, IntelliJ's code cleanup will delete the import because it believes

it is unused.

- In some IntelliJ versions, the code that removes the green background overrides the yellow tint that appears on the line where the cursor is.
- Assigning a value to a method parameter within a JML specification gives the incorrect error message "Variable expected". Since assigning variables is not allowed in general, it is correct that there is an error, only the message is wrong.
- Pure checking is insufficient: It does not check if variables are assigned inside the method, and marking a void method as pure is possible.
- No message is given when `modifiable` clauses conflict, such as having `modifiable \everything` and `modifiable \nothing` in the same JML comment.
- Pressing enter inside an incomplete JML comment adds a spurious `*/` under certain conditions.
- Copying and pasting a JML specification does not automatically correct indentation, unlike copying and pasting Java.
- Auto-formatting the code may result in awkward indentation in JML comments under certain conditions.
- Using Ctrl+/ inside a JML comment adds `//` to the start of the line, breaking the JML lexer. JML comments of the form `(* *)` should be used instead.
- Generics are not properly type checked. For example, the class name of the generic does not need to exist.
- Code completion does not work on JML backslash expressions related to types.

References

1. Jemerov, D. *IntelliJ SDK: publish javadoc online* 22nd February 2019. https://youtrack.jetbrains.com/issue/IJSDK-19.
2. JetBrains s.r.o. *IntelliJ Platform SDK* 15th March 2021. https://plugins.jetbrains.com/docs/intellij/welcome.html.
3. Ellis, M. *IntelliJ project migrates to Java 11* 18th September 2020. https://blog.jetbrains.com/platform/2020/09/intellij-project-migrates-to-java-11/.