Parsing and Printing Java 7-15 by Extending an Existing Metamodel

Martin Armbruster
Supervisor: Manar Mazkatli

July 28, 2022

Many technologies and frameworks are built upon the open source Eclipse Modelling Framework (EMF) to provide model-based software development or even model-based consistency preservation of software artifacts. In this context, not only EMF-based modeling of the source code but also parsing of the source code and printing the model again into source code files are required.

The Java Model Parser and Printer (JaMoPP) provides an EMF-based environment for modeling, parsing and printing Java source code [16]. However, it supports just the syntax of Java 5 and 6. Moreover, JaMoPP is based on some technologies that have technical problems and have not been further maintained.

In this work, we extend the metamodel of JaMoPP to support Java versions 7-15. Our extensions expand the metamodel with new features, for instance, the diamond operator, lambda expressions, or modules. Moreover, we implemented our new parser and printer. The parser implementation is based on the Eclipse Java Development Tools (JDT) that is well maintained, which reduces the maintenance effort to extend our JaMoPP for new versions of Java.

1 Introduction and Foundation

The Java Model Parser and Printer (JaMoPP) provides an Eclipse Modeling Framework (EMF)-based environment for modeling Java source code [16]. Therefore, JaMoPP contains an Ecore-based Java meta-model conforming to *The Java Language Specification - Third Edition* (JLS 3) [16] specifying the syntax of Java 5 and 6 [15, 20]. JaMoPP also defines the Java syntax in the CS specification language of EMFText [16]. Out of the CS specification, EMFText generates code for two artefacts [17]. At first, a parser based on ANTLR is generated to create models from source code files, and, secondly, a pretty printer writes source code files from models [17, 16].

In order to establish the connections between different Java models introduced by, e. g., imports, JaMoPP includes a mechanism to resolve such references [16]. In detail, the references contain proxy objects that are resolved to the actual objects based on the proxy object URI when a reference is accessed [19]. EMFText already generates a set of resolvers with a default implementation [17] which are integrated into EMF's proxy resolution mechanism [18]. Thus, JaMoPP extends the resolvers with respect to Java-specific properties [18].

The architecture of JaMoPP in the context of EMF is depicted in Figure 1. In EMF, a model is contained within a `Resource` (1) which is contained in a `ResourceSet` (6) [18]. Reversely,

a `ResourceSet` manages `Resources` whereby it relies on `ResourceFactories` (8) to create `Resources`. As mentioned earlier, EMFText generates a `JavaParser` (3) and a `JavaPrinter` (4) hidden behind a specific `JavaResource` (2) which is created by a `JavaResourceFactory` (9). The generated code is supplemented by a `ClassFileModelLoader` (5) which generates models from class files using the Byte Code Engineering Library (Apache Commons BCEL). As a result, modelling tools (7) can put Java source or class files into EMF which get loaded by a `JavaResource`. The Java-specific resolvers (10) establish the references between model elements. As the targets of references are spread across several different `Resources`, the resolvers can also load models on demand. Because the resolvers do not know the physical location of Java files, they use a unique location-independent URI for the model elements. JaMoPP provides the `Classpath` (12) for connecting the `Resources` by generating the unique URIs hiding the physical location and storing a mapping from the unique URI to the physical location in the `URIMap` (11) from which `ResourceFactories` can retrieve the physical location.

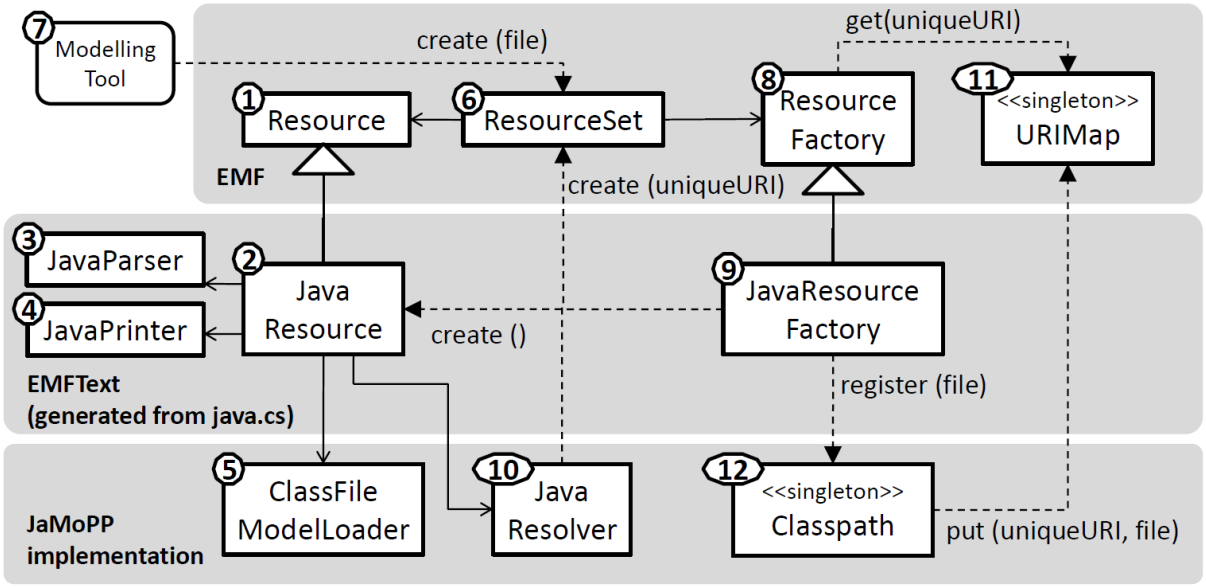

Figure 1: Architecture of JaMoPP [18].

For validating and evaluating JaMoPP, the following testing process depicted in Figure 2 was executed [18]. A Java source file (shown in the upper left corner) is parsed into a JaMoPP model. Its references are resolved and de-resolved to test the reference resolution and to print the model. Both the original and reprinted source files are parsed with the Eclipse Java Development Tools (JDT) Core into abstract syntax trees (ASTs) [18, 31]. Then, the ASTs are compared with the JDT AST Matcher to check if they are equal [18]. In this case, the test is successful.

In the remainder, the previously described JaMoPP version will be mentioned as *the original JaMoPP version*. Due to the limitation of JaMoPP to Java 6 [16] and the availability of Java 17 [26], the adaptation and extension of the original JaMoPP version is covered here. This includes the extension of the meta-model in section 2 and three implemented variants in section 3, section 4, and section 5 for the reference resolution. A comparison of all three variants in section 6 complements their description. After potential future work presented in section 7, this documentation is concluded in section 8.

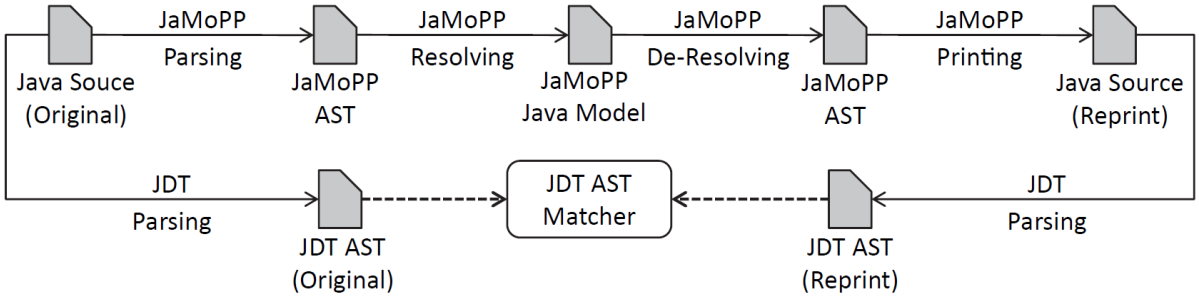

Figure 2: Test process for JaMoPP [18].

2 The JaMoPP Meta-Model Extension

To support the features of newer Java versions, the JaMoPP meta-model was extended. In the following, the features and their model representation are presented.

Diamond in Class Instance Creation Expressions A simple example is `new ArrayList<>()` for the diamond which was added in the JLS 7 [12]. It is represented by the class `java.instantiations.NewConstructorCallWithInferredTypeArguments` which is a subclass of `java.instantiations.NewConstructorCall`.

`try`-with-resources Statement To represent, for instance, `try (someField; FileReader in = new FileReader("SomeFile.txt"))` added in the JLS 7 [12] in the state of the JLS 16 [10], the `java.statements.TryBlock` was extended with references to `java.variables.Resource` where `java.variables.LocalVariable` and `java.references.ElementReference` inherit from `java.variables.Resource`.

Multi-`catch` Multi-catch, i. e., e. g., `catch (IOException | IllegalArgumentException e)`, introduced in the JLS 7 [12] is realized by setting an instance of the `java.parameters.CatchParameter` class as the parameter of a `java.statements.CatchBlock`. `java.parameters.CatchParameter` is a specialization of `java.parameters.OrdinaryParameter` and contains a list of additional type references.

Binary Integer Literals Binary integer literals such as `0b1111` from the JLS 7 [12] are modeled by the `java.literals.BinaryIntegerLiteral` and `java.literals.BinaryLongLiteral` classes.

Lambda Expressions With the JLS 8, lambda expressions were introduced [13]. Their modeling is presented in Figure 3. A `java.expressions.LambdaExpression` is a special `java.expressions.Expression` consisting of `java.expressions.LambdaParameters` and a `java.expressions.LambdaBody`. The parameters can be explicitly or implicitly typed. For instance, the lambda expression `(Object o) -> {}` is explicitly typed

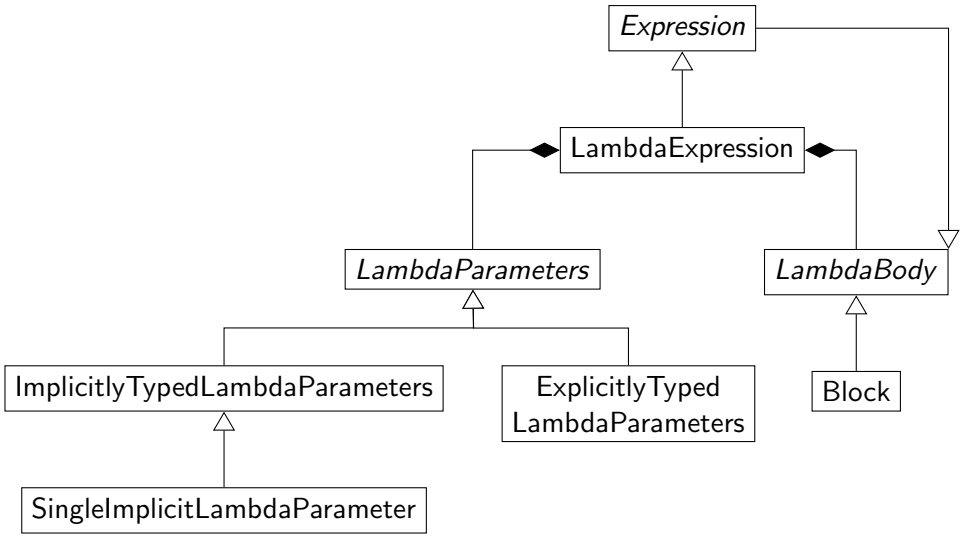

Figure 3: Modeling of lambda expressions.

whereas `(i, j) -> i + j` and `i -> i` are implicitly typed. Both forms are represented by the `java.expressions.ExplicitlyTypedLambdaParameters` and `java.expressions.ImplicitlyTypedLambdaParameters` with the special `java.expressions.SingleImplicitLambdaParameter` for the case that an implicitly typed lambda parameter is given without parentheses. The body of a lambda expression is an expression or a block so that both classes inherit from `java.expressions.LambdaBody`.

Method Reference Expressions Method reference expressions as `Object::toString` were added in the JLS 8 [13] and have a corresponding abstract `java.expressions.MethodReferenceExpression` class in the meta-model which is integrated into the expression class hierarchy. An excerpt of the hierarchy for the method reference expressions is shown in Figure 4. `java.expressions.MethodReferenceExpressions` are differentiated in `java.expressions.ClassTypeConstructorReferenceExpressions` to cover, for instance, `String::new`, `java.expressions.ArrayConstructorReferenceExpressions` to cover, e. g., `int[]::new`, and `java.expressions.PrimaryExpressionReferenceExpression`. The latter class contains a mandatory reference to a `java.expressions.MethodReferenceExpressionChild` and an optional reference to a method.

Receiver Parameters For receiver parameters added in the JLS 8 [13], the `java.parameters.ReceiverParameter` as a special `java.parameters.Parameter` was added.

Default Interface Methods Beginning with the JLS 8, interface methods can be declared `default` and can have an implementation [13]. Therefore, while it is still distinguished between `java.members.ClassMethods` and `java.members.InterfaceMethods`, their superclass `java.members.Method` instead of the `java.members.ClassMethod` contains a reference to a `java.statements.Statement` representing the body of the method. Additionally, the modifier `java.modifiers.Default` has been added to model the `default` modifier.

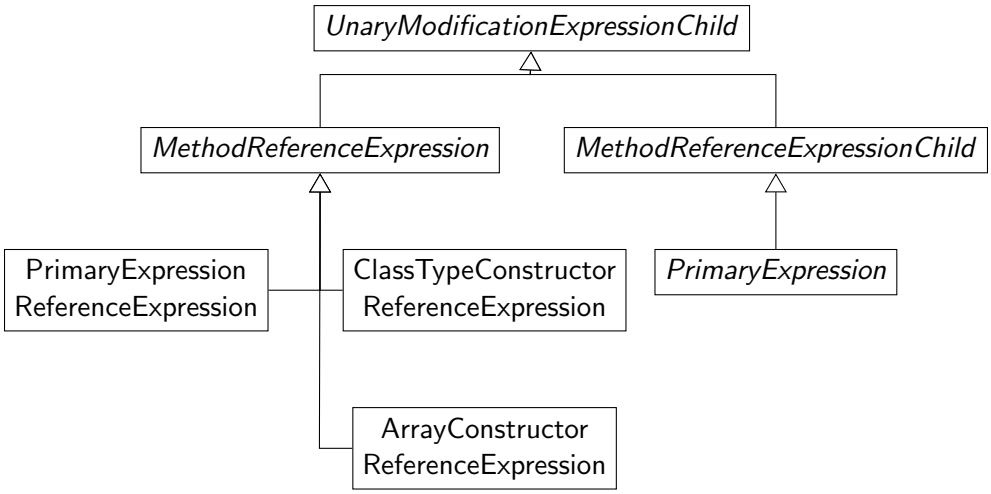

Figure 4: Modeling of method reference expressions.

Further Positions of Annotations In the JLS 8, further positions for annotations were added [13]. As a result, the corresponding positions in the meta-model were extended to support the annotations.

Modules In the JLS 9, module declarations were added [14]. Their model representation is depicted in Figure 5. A `java.modules.Module` is a new `java.containers.JavaRoot` and contains `java.modules.ModuleDirectives`. A `java.modules.ModuleDirective` is a `java.modules.RequiresModuleDirective`, `java.modules.UsesModuleDirective`, `java.modules.ProvidesModuleDirective`, `java.modules.OpensModuleDirective`, or `java.modules.ExportsModuleDirective`. Every concrete subclass has references to the types, packages, or modules it describes.

`var` as Local Variable Type The JLS 10 introduced `var` as a possible type in local variable declarations [4] while the JLS 11 extended the use of `var` for explicitly typed lambda parameters [5]. In every case, the actual type is inferred from the context. Thus, the `java.types.InferableType` as a subclass of `java.types.TypeReference` models the use of `var`. It also includes a list of `java.types.TypeReferences` which can store the actual types.

Switch Expressions The `java.statements.Switch` extends the `java.expressions.UnaryModificationExpressionChild` to realize switch expressions introduced with the JLS 14 [8]. In addition, to model switch rules, for instance, `case 2, 3 -> 2;`, the subclass `java.statements.SwitchRule` for the `java.statements.SwitchCase` was added. It differentiates between a `java.statements.DefaultSwitchRule` and `java.statements.NormalSwitchRule`. The `java.statements.NormalSwitchRule` and `java.statements.NormalSwitchCase` contain an additional list of expressions for further case constants. At last, the `java.statements.YieldStatement` complements the switch expression modeling.

Text Blocks A text block from the JLS 15 [9] is represented by the `java.references.TextBlockReference` class as the subclass of `java.references.Reference`

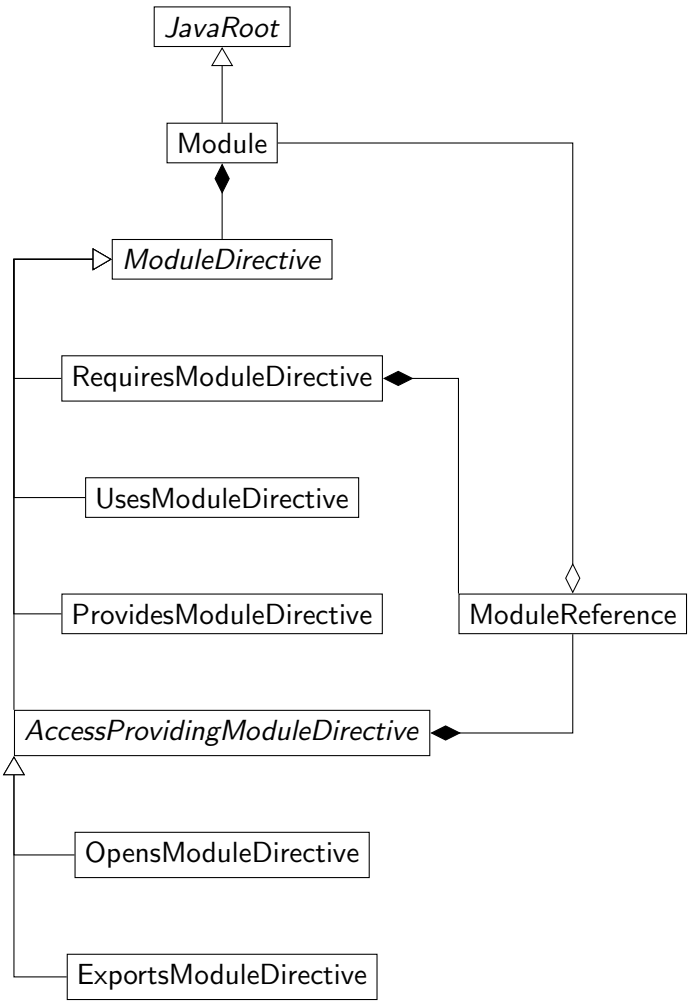

Figure 5: Modeling of modules.

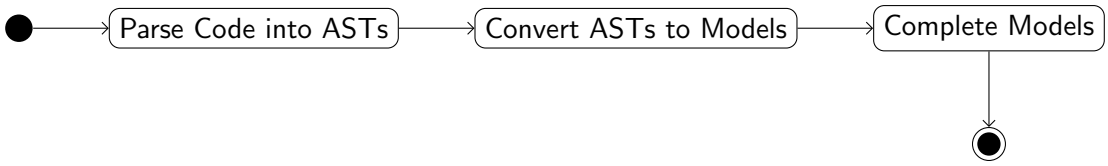

Figure 6: Overview over the first variant's parsing process.

containing the value of the text block.

Additional Changes Beside the aforementioned additions, further elements were added to or changed in the meta-model. Every `java.containers.JavaRoot` stores its origin which can be a source file, class file, a file from an archive, or a binding. Moreover, `java.types.TypeReference` inherits from `java.arrays.ArrayTypeable` so that every occurrence of a type can directly store its array dimensions and the information if it is an array type. As a consequence, array types can also be modeled as type arguments which was not supported before, and most inheritances of `java.arrays.ArrayTypeable` were removed. At last, a `java.references.Reference` includes an optional list of `java.types.TypeReferences` to store its concrete type with potential concrete type arguments. This allows the explicit representation of types in contexts in which the types are usually inferred.

Final Remarks Records, patterns, and the pattern matching for `instanceof` which are the new features in the JLS 16 [10], and sealed classes introduced in the JLS 17 [11] have currently no model representation.

At first, the CS specification of the original JaMoPP version was extended to include the new features of Java. However, this caused several problems of which only some parts could be solved and which limit the future development. Therefore, the first variant was implemented.

3 First Variant: Using the Eclipse JDT Core's Bindings

In the historical established first variant, the dependency to EMFText was removed by replacing the CS specification and generated parser and printer implementations with manual implementations.

The manual printer implementation consists of a single class which is responsible for outputting the textual representation of every model element independently of the parser. The generated Java source code has valid syntax and preserves the semantics, but neither the layout nor the documentation.

The manual parser implementation uses the Eclipse JDT Core to transform source code files into an AST which is converted to the actual model. Figure 6 gives an overview over the first variant's parsing process.

The following assumption is central for this variant:

A1 The complete source code and every dependency is available.

The source code including its dependencies is parsed with the Eclipse JDT Core to obtain the ASTs. All generated ASTs are converted to JaMoPP model instances. During the conversion, references within a model instance to separated model elements are directly set without creating proxy objects. The `ASTParser` of the Eclipse JDT Core resolves references between source code files and ASTs to bindings [1]. A binding represents a package, module, type, annotation, member value pair in an annotation, variable, or method [22]. Therefore, a binding can be used to identify the corresponding model element and to directly set a reference. Furthermore, bindings contain all information of the corresponding Java element [22]. A type binding, for instance, includes bindings to the declared fields, methods, and inner types [24]. This means that bindings can be converted to models if there is no corresponding source file. As a consequence of using the bindings to resolve references, the reference resolution mechanism of the original JaMoPP version including the generation of proxy objects was removed.

```
1  class A {
2          void m() {
3          }
4  }
5  class B {
6          A b;
7          A c;
8          A d = new A() {
9                  @Override
10                 void m() {
11                         s();
12                 }
13
14                 private void s() {
15                 }
16         };
17 }
```

Listing 1: A source code example for bindings in the first variant.

Considering the source code example in Listing 1, the AST nodes for the fields of class B return type bindings representing class A. During the conversion of B's AST, these type bindings are used to obtain the model of A. To ensure that there is only one model for A, a mapping between the binding and the model of A is stored in the class `JDTResolverUtility`. If the model for A is requested for the first time, it will be created and stored in the mapping. If the model for A is requested afterwards, the created model is returned from the mapping.

The actual mapping between a binding and a model element consists of a mapping function for bindings to String names and a mapping from a String name to the model element. Most of the time, the composition of the name depends on the type of the binding and is independent of the concrete binding object. For the fields b and c in the source code example, two different type binding objects are returned. Nevertheless, both type bindings represent class A. Thus, binding objects cannot be directly mapped to model elements, and a specific name has to be used.

After the conversion, a completion step is performed. For references into the JDK or dependencies, only the referenced model elements are created without connecting them. As a consequence, this last step completes these elements. Classes that are only represented as bindings are converted to model instances to establish the connections between existing elements. As the converted bindings can reference further parts within the JDK or dependencies, the completion step is executed until

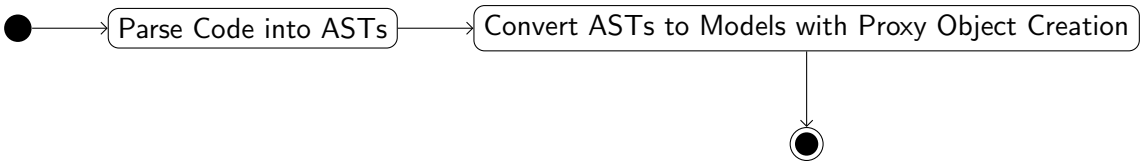

Figure 7: Overview over the parsing process of the second variant.

there are no more bindings to convert or elements to connect.

With the end of the completion step, the parsing ends and all model instances are complete and valid.

The first variant has a second implicit assumption:

A2 Every package and module has a model representation.

According to the JLS 3 and 7-17, every compilation unit is contained within a package [15, 12, 13, 14, 4, 5, 6, 7, 8, 9, 10, 11]. Analogously, according to the JLS 9-17, every package is included in a module [14, 4, 5, 6, 7, 8, 9, 10, 11]. This circumstance is reflected in the bindings: a type binding has a method for obtaining its package binding [24], and a package binding has a method for getting the module binding [23]. All of these bindings are converted to models at the latest during the completion step so that every package and module has a model representation. As a consequence, in the JaMoPP meta-model, `ConcreteClassifiers` contain and require a reference to a package, and a package references its containing module.

The first variant has one limitation (**L1**): If there are changes within the source code, the source code and its dependencies have to be completely parsed again because there is no mechanism to parse single files or to exchange model instances.

4 Second Variant: Re-introducing Proxy Objects

To overcome the limitation **L1** of the first variant, the second variant was implemented. It re-introduces and adapts the proxy object creation and resolution mechanism of the original JaMoPP version including the `ClassFileModelLoader`. Figure 7 shows the parsing process of the second variant. Compared to the first variant, the assumption **A1** has not to be fulfilled.

One or more Java files are parsed with the Eclipse JDT Core to obtain their ASTs. The bindings are ignored so that the ASTs are only converted to model instances. For the references to other model elements, proxy objects are created similar to the original JaMoPP version. This leads to the generation of a unique URI that identifies the proxy object and its context which is used to find the actual model element at a later point in time. After the conversion of the ASTs, the parsing ends.

Considering the source code example in Listing 1, the modeled fields of class B contain no direct reference to the model of A after the parsing. Instead, they contain only proxy objects.

If the actual model element of a reference is needed, the reference resolution of the original JaMoPP version is executed. Starting with the current container element of the proxy object, the resolution mechanism walks the model hierarchy upwards looking into the model elements the mechanism

encounters [25]. If a suitable model element is found, the proxy object is replaced with the found model element [19]. If the resolution mechanism reaches the top of the model hierarchy, it continues in models referenced by, e. g., imports [25]. If no model element is found, the proxy object is returned as the result of the resolution [19].

The resolution mechanism is extended to find elements in the newly added classes and features of the meta-model. For example, `LambdaParameters` are considered. Herein, a potential limitation (**L2**) of the second variant arises. The `MethodDecider` responsible for finding a method to a method call relies on a comparison of the method and method call which includes the parameter and argument types [25]. As the arguments are expressions [10, 25], their type is calculated in the `ExpressionExtension`, `ReferenceExtension`, and `TypeParameterExtension` if type parameters are involved [25]. The introduction of lambda expressions, method reference expressions, and `var` required the extension of the type calculation because their type calculation depends on the surrounding context [10]. For example, in the variable declaration `Function<Integer, Integer> func = i -> i;`, the types of `i` and of the lambda expression are inferred from the type `Function<Integer, Integer>` of the local variable. Another example related to the `MethodDecider` is `someMethod(i -> i)`. To find the type of `i` in the context of the extended type calculation in JaMoPP, the type of the lambda expression has to be known. To find the lambda expression's type, the parameter type of `someMethod` is considered which requires the resolution of the method call. During the resolution, the argument type, i. e., the type of the lambda expression, would be compared to types of potential methods [25] as mentioned before. However, the argument type calculation, i. e., the type calculation for the lambda expression, would require the resolution of the method call forming an endless loop. Therefore, the type calculation has been extended to avoid such loops by, e. g., temporarily assigning an unkown type to lambda expressions in method calls and to avoid other conflicts. Nonetheless, it is unclear if all cases are covered and if the current architecture for calculating types and type parameters is sufficient for future enhancements (**L2**).

With the separation of the parsing and reference resolution, the assumption **A1** has not to be fullfilled if references to non-available files are not resolved. However, as the bindings are ignored, model instances of packages and modules are only generated if there are corresponding `package-info.java` and `module-info.java` files. As a consequence, assumption **A2** does not hold. Therefore, the metamodel was adapted so that the reference of `ConcreteClassifiers` to packages and the reference of packages to modules are optional. Furthermore, the `java.references.PackageReference` represents packages in expressions, e. g., `java` and `java.lang` in `r = java.lang.Object.class;`.

The `Classpath` from the original JaMoPP version, actually the `JavaClasspath` [25], was not removed. In the first variant, it provides access to all models to establish references. With the re-introduction of the reference resolution and proxy object creation in the second variant, the `JavaClasspath` was employed to store the mapping between logical URIs and the physical location of files again. Initially, in the original JaMoPP version, the `JavaClasspath` distinguished between different `ResourceSets` and a global scope for the mapping [25]. While there is no such differentiation in the first variant, it was re-implemented in the second variant.

To increase the maintainability of the manual printer implementation of the first variant, the printer was split into multiple classes.

5 Third Variant: Combination of the First and Second Variant

The third variant extends the principle of the second variant by incorporating a binding-based resolution which is similar to the first variant.

The third variant's parsing process is shown in Figure 8. It starts with the parsing of the source code by the Eclipse JDT Core. Afterwards, the resulting ASTs are converted to models. Hereby, proxy objects are created for the references as in the second variant. However, compared to the second variant, the bindings are stored in the context of a proxy object so that they are used directly after the conversion to obtain the model elements and to resolve references. If all bindings have been investigated, the parsing ends. If there are unresolved references and their resolution is requested at a later point in time, the resolution mechanism of the second variant is used.

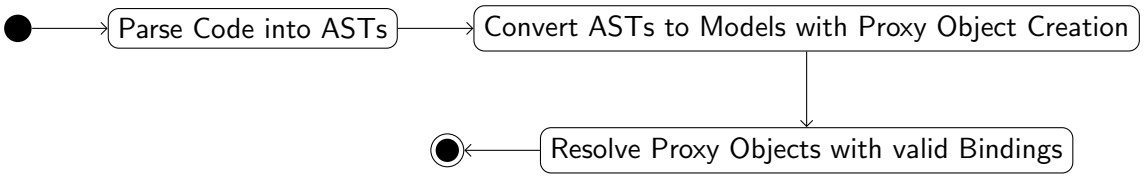

Figure 8: Parsing process of the third variant.

Parts of the architecture and organization of the binding-based resolution is shown in Figure 9. While every binding type has its own specialized class for the resolution, the resolution process is hierarchically organized. To resolve methods or fields, for example, their declaring type is resolved beforehand. If a type is also contained within another type, the other type is resolved at first. This procedure is repeated until a top level type is reached. If the top level type is part of the parsed source code, its model is available and can be directly returned by finding the model through its unique fully qualified name. In the case of top level types in dependencies or the JDK, there is only the binding. Thus, similar to the first variant, the binding is converted to a model so that all contained types, methods, and fields have a model representation. After a top level type has been resolved, it is used to obtain the method, field, or nested type.

The process to find inner and anonymous classes is different compared to the search for other elements because they have no name or no unique name which can be used to identify them. Although the bindings provide a method to obtain a unique key for a binding [22], it cannot be used to find the corresponding model element. However, type bindings also have a method to get the binary name

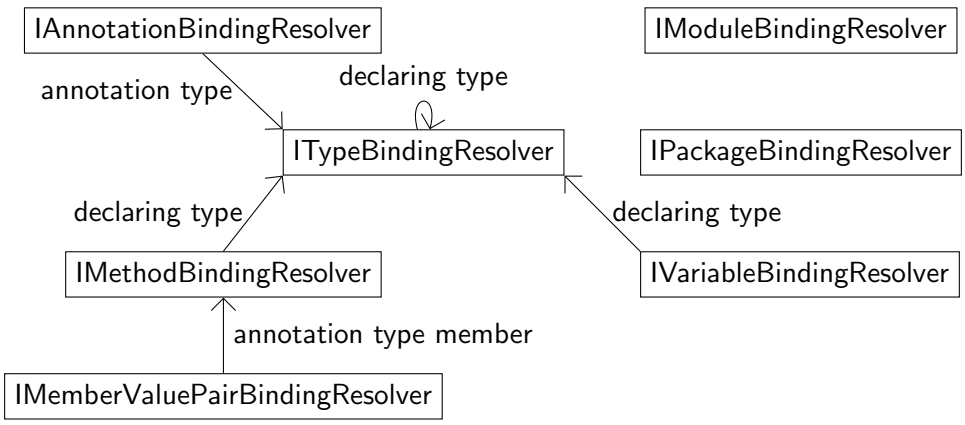

Figure 9: Organization of the binding-based resolution.

of a type as it is defined in the JLS 3 [24]. According to the definition,

> "The binary name [...] consists of the binary name of its immediately enclosing type, followed by $, followed by a non-empty sequence of digits, followed by the simple name of the local class" [15]

in the case of a local class. Practically, the digits form a number which gets incremented for every inner or anonymous class within a type. It is distinguished between numbers for innner and anonymous classes and numbers for different hierarchical levels. For example, both anonymous classes in the source code example in Listing 2 have the same number 0 while the first inner class `C` gets the number 0 and the second inner class `C` gets the number 1. Using the number, it is possible to find an inner or anonymous class by counting the classes within the declaring type. If the counted and provided number are equal, the inner or anonymous class is found.

```
1  class Z {}
2  class A {
3          class B {
4                  public void m() {
5                          Z z = new Z() {
6                                  public void n() {
7                                          Z y = new Z() {};
8                                  }
9                          };
10                         { class C {} }
11                         { class C {} }
12                 }
13         }
14 }
```

Listing 2: A source code example for the resolution of inner and anonymous classes.

In the case of local variables or parameters, their bindings are not used for the resolution because the implementation effort exceeds the benefit as the resolution mechanism of the second variant is already implemented and sufficient for the resolution.

The Eclipse JDT Core infers types for local variable declarations or lambda parameters with `var` as type in form of type bindings [2, 3]. As a consequence, the third variant converts such type bindings to type references and sets them as the actual types of `InferableTypes`.

If the third variant is used without the binding-based resolution, the second variant is effectively used. Therefore, the third variant allows the control of the binding-based resolution with specific parser options. The option RESOLVE_BINDINGS enables the binding-based resolution at all. Next, the option RESOLVE_BINDINGS_OF_INFERABLE_TYPES controls whether type bindings of inferred types are converted. With the option RESOLVE_ALL_BINDINGS, the binding-based resolution is repeatedly performed until there are no bindings left similar to the completion step of the first variant. Because the models contain proxy objects even after a full binding-based resolution, the option RESOLVE_EVERYTHING resolves all remaining proxy objects with the second variant so that the models after the parsing only include proxy objects which cannot be resolved. While the second variant loads files on demand to resolve references, the third variant can convert bindings instead of loading the corresponding files. As a result, the option PREFER_BINDING_CONVERSION allows to control if bindings shall be converted or files shall be loaded. In addition, the option CREATE_LAYOUT_INFORMATION prevents the generation of layout information if it is disabled,

and the option REGISTER_LOCAL enables or disables the global or local, i. e., for a specific `ResourceSet`, storage of the mappings in the `JavaClasspath`.

Regarding the output of the Java models in the *XML Metadata Interchange* (XMI [30]) format, all three variants support the format. Additionally, the new `JavaResource2` provides the possibility to save Java models as `javaxmi` files directly in the XMI format. If only specific models of all generated models are stored in the XMI format and they are programmatically loaded again or opened in the XMI editor, errors arise because EMF cannot find the non-stored models referenced by the stored models. In case of the second and third variant, proxy objects are stored in the XMI format, but also cause errors when they are accessed. Thus, a temporary solution is to resolve all references and to store all models.

Similar to the second variant, the assumption **A1** has not be fullfilled if references to non-existing model elements are not resolved, and the assumption **A2** does not hold. The limitations **L2** remains.

6 Comparison of the Three Variants

JaMoPP offers three variants for the reference resolution and multiple parser options in the third variant. As a consequence, the selection of a suitable variant depends on the use case and its goals. Therefore, the variants are compared in the following to guide the selection.

Due to the architectural design of JaMoPP, all variants differentiate between the source code of a project which is parsed with the Eclipse JDT Core and the dependencies of a project and the JDK which are usually not available in their source code form and represented as bindings.

To improve the insight into the implementation of all variants, their execution time was measured. On a computer with an Intel Core i7-4790K CPU and 32 GB working memory, JaMoPP parsed the TeaStore[1] in version 1.4.0. The TeaStore provides a Web-based store for Tea and related products [27]. It is designed as a test and benchmarking framework for the evaluation of, e. g., performance modeling approaches, run-time auto-scalers, or energy efficiency and power prediction methods and is implemented in Java. Before the execution of JaMoPP, the TeaStore was built so that all dependencies were available and all proxy objects could be resolved. For the execution, the first variant[2] and the third variant with varying parser option settings[3] were selected and abbreviated with the following letters:

FV: First Variant.

SV: Second Variant which is the third variant where all resolution options are turned off.

OL: One level of the resolution is executed in the third variant, i. e., only the options RESOLVE_BINDINGS and RESOLVE_BINDINGS_OF_INFERABLE_TYPES are turned on.

WR: The third variant without the RESOLVE_EVERYTHING option turned on is executed.

FR: Full resolution in which the third variant has all resolution options turned on.

[1] https://github.com/DescartesResearch/TeaStore
[2] Corresponding to commit `94bed8fd94b52d354473df0d667d1ffb70ae265b` in the JaMoPP repository.
[3] Corresponding to commit `7c980cd1bf1bd381dc1a85577011df72c123f2bc` in the JaMoPP repository.

For the execution of the third variant, the parser options CREATE_LAYOUT_INFORMATION and REGISTER_LOCAL were always turned on. In order to improve the comparability between the executions, the goal of the parsing includes a full resolution of all references. In case of **SV**, **OL**, and **WR**, the process is split into the parsing phase and a resolution phase in which all remaining proxy objects are resolved. However, in the resolution phase of **SV** and **OL**, the time limit of one hour was exceeded. Thus, in **SV** and **OL**, only the proxy objects in the source code models were resolved. In addition, all measurements were executed 20 times except for **FR** which was repeated 100 times. Beside the parsing, the complete test process of the original JaMoPP version was executed to ensure that the parsed models are correctly parsed and printed.

Table 1 displays the measured average execution times. The first variant completes in 13.6s on average. In comparison to the combined execution times of the other variants, it is the fastest variant. The combined execution times for the second and third variant are higher because the parsing and resolution are separated by employing proxy objects which add an overhead and because the dependency and JDK models in **SV** and **OL** contain more proxy objects compared to **WR** and **FR** which convert more bindings to models. Considering the second variant, it has the smallest parsing time with 3.4s because it only converts the ASTs to models. The more resolution is performed after the parsing, the longer the parsing process takes to finish.

FV	SV		OL		WR		FR
	Parsing	Resolution	Parsing	Resolution	Parsing	Resolution	
13.6s	3.4s	72.1s	6.6s	129.3s	105.1s	3.4s	128s

Table 1: Average execution times for JaMoPP with different variants and parser options.

As previously mentioned, all three variants can provide complete models. While the first variant and third variant with all resolution options turned on provide the complete models directly after the parsing, the second and third variant decouple the resolution from the parsing and allow the resolution on demand. As a consequence, it needs to be considered which models are accessed and when the access takes place. If the models of the dependencies or the JDK are not frequently accessed or not at all, proxy objects with a partwise resolution can be appropriate. Additionally, the usage of proxy objects saves memory. For **FV** and **FR**, all models were also stored in the XMI format to compare the size of all models to the size of the source code models. In **FV**, 4718 models need 89.7MB space from which 282 files contain the source code models of the TeaStore including its test files and taking 10.6MB of space (12% compared to all models). In comparison, **FR** produces 3921 model files with overall 102MB and 201 source code models with 14.1MB (14% compared to all models). The derivation between **FV** and **FR** comes from the exclusion of the test files and changes in the meta-model.

As outlined in section 3 with limitation **L1**, changes in the source code require a complete parsing of the code in the first variant while only the changed files can be parsed in the second and third variant.

Considering the internal implementation of JaMoPP, it is expected that all three variants require approximately the same maintenance effort. In the first variant, the parsing and resolution are tightly coupled and respect several edge cases. Therefore, extensions can be complex to be implemented. In contrast, the second and third variant base their resolution on the architecture of the original JaMoPP version with limitation **L2** and including type calculations. Here, enhancements can require adjustments in the architecture or type calculation.

7 Potential Future Work

This section covers topics for possible future enhancements and improvements of JaMoPP.

7.1 Support for Further Java Versions

Currently, Java files conforming to the JLS 3 and 6-15 can be represented as models of the JaMoPP meta-model. Based on the JLS 16 and 17, their new features can be implemented. It requires extensions at different locations in the source code. At first, the meta-model needs to be enhanced to contain corresponding model representations for the features. Afterwards, the parser and printer are extended to generate and output the model elements. At last, it needs to be considered which changes take place in the reference resolution. Depending on the features, the resolution has to check additional model elements as the potential target of a reference. For example, records or local variables as the pattern in `instanceof` expressions are such possible targets. Furthermore, new features can introduce new locations for proxy objects or new types of references which can require further or extended resolvers.

7.2 Updating the First Variant's Parser and Unifying the Parser Implementations

As hinted in the comparison of the three variants in section 6 and due to the separated development of the first and second / third variant, there is one parser for the first variant and one parser for the second / third variant. Both parsers diverged from each other during the development so that the parser of the first parser is not up-to-date with the latest changes in the meta-model and architecture introduced by the second / third variant. Therefore, future work can include the update of the first variant.

In order to update the first variant's parser, there are two possibilities. At first, the implemented parser can be adapted and updated to cover the changes. On the other hand, both parsers can be combined to reduce the maintenance effort by having only one parser. For the combination, two interfaces are required: a model element provider and a reference provider interface. The model element provider injects the correct model objects into the parser. In the first variant, the model element provider handles the model objects as described in section 3. As a result, model objects are created and cached on their first retrieval and returned in subsequent requests. The implementation for the second / third variant always creates new model objects. The reference provider interface is responsible for handling the references. While it looks up the target and sets references directly in the first variant, the second / third variant's implementation creates proxy objects. The `JDTResolverUtility` with the completion step remains for the first variant.

7.3 Improve Handling of `$` in Type Names

According to the JLS 3 and 7-17 [15, 12, 13, 14, 4, 5, 6, 7, 8, 9, 10, 11], type names are allowed to contain `$` characters. At the same time, the binary name of inner and anonymous types include a `$` to separate the binary name of their declaring type from the inner or anonymous type's name. By using the `ClassFileModelLoader` in the second and third variant, JaMoPP differentiates between `.` and `$` as separators for packages and types in fully qualified names in certain places to support

both meanings of $. As a consequence, it can be checked if the `ClassFileModelLoader` allows to identify all individual type names of a binary name including $ characters in type names and omitting them as name separators. If this determination is possible, the code for handling $ in type names can be simplified and reduces the maintenance effort.

7.4 Adopting Parts of Xtext

Xtext provides an environment to define grammars for domain-specific languages [28]. It creates an Ecore-based meta-model, a parser to generate models, and a serializer to print the models back into the textual syntax [21]. As the models are integrated into EMF, Xtext also includes a mechanism to create proxy objects for references [29]. A proxy object URI encodes the proxy object's context and is used to find the referenced element by a linking service. Xtext allows language-dependent implementations of the linking service and extensions of the default simple linking service implementation.

By comparing the mechanisms of Xtext and JaMoPP, similarities between both environments regarding the proxy object URI and reference resolution are found. Therefore, it can be checked which parts of Xtext can be reused or adopted in order to reduce the complexity and maintenance effort of JaMoPP.

Currently, the context of proxy objects in JaMoPP are separatedly stored in in-memory objects which are not persisted. If the proxy object URI generation of Xtext can be applied on JaMoPP's proxy objects, the additional context object is not needed. Moreover, the context is persisted and can be restored from the proxy object URI. At the same time, the removal of the context object requires adaptations to the reference resolution and a new mapping between the proxy objects and their corresponding bindings for the binding-based resolution.

Combining the reference resolution of JaMoPP with Xtext's linking service, JaMoPP's reference resolution can become a specialization of Xtext's linking service. It can form an architecture which easily allows future extensions and overcomes **L2**. However, a switch to the linking service requires major changes in JaMoPP's reference resolution and thus implementation effort.

7.5 Recovery Mechanisms

As mentioned before, proxy objects which cannot be resolved cause issues if they are accessed. Thus, a recovery mechanism can mitigate connected errors by creating model elements for unresolvable proxy objects. It assumes that the reference resolution has been executed before. The actual goal and complexity of a recovery mechanism depends on the concrete use case. A simple recovery mechanism as example can provide only model elements to prevent having proxy objects without generating complete corresponding models. If more accurate models are required, an advanced recovery mechanism needs to be implemented. In the following, a simple and an advanced recovery mechanism are proposed. It is their first proposal so that they can be incomplete and can potentially not cover all cases due to the complexity of the Java syntax.

Both mechanisms consider all types of proxy objects and container of proxy objects. This includes import statements, annotations, methods for annotation values, references to modules, references to packages, references to types (represented by `TypeReferences`), and references in expressions which are further divided into constructor calls, method calls and identifier references. In the latter context,

an identifier can reference a package, type, field, local variable, method parameter or method (in a method reference expression). As the printer uses the name of proxy objects of the resolved elements to output the name, the recovery mechanisms ensure that the generated elements have a suitable name.

7.5.1 A Simple Recovery Mechanism

The simple recovery mechanism aims at generating model elements for proxy objects to avoid having proxy objects while the elements are loosely connected to have valid models. Thus, before the actual element is recovered, in an initialization step, one `JavaResource` is created which will contain all root model elements generated in the following. Beside packages and modules as root elements, only one `CompilationUnit` is required which will hold all types. In addition, a container class with an arbitrary name for all new fields, constructors, and methods is created.

Import Statements There are four types of import statements [9]:

1. single-type-import, e. g., `import java.lang.Object;`,
2. type-import-on-demand, e. g., `import java.lang.*;`,
3. single-static-import, e. g., `import static org.junit.Assert.assertTrue;`, and
4. static-import-on-demand, e. g., `import static org.junit.Assert.*;`.

Sinlge-type-imports and static-imports-on-demand refer to exactly one type [9] so that a class with the corresponding name can be created. Nonetheless, it has to be considered that the original imported type can be the inner type of another type as shown in the example Listing 3. Although the JLS suggests a naming convention in which type names start with uppercase letters and packages with lowercase letters [9] which would allow to differentiate types from packages, the alternative implementation of class B depicted in Listing 4 is also valid code and compiles. Therefore, the simple recovery mechanism always assumes packages for all parts of the qualified name before the type name and does not consider potential outer classes. This strategy (**S1**) will be applied throughout both recovery mechanisms.

```
1  package x.y;
2
3  public class A {
4          public static class B {
5                  public static void m() {}
6          }
7  }
8
9  package k.l;
10
11 import static x.y.A.B.*;
12
13 public class C {
14 }
```

Listing 3: Source code example for the import of an inner class.

```
1  package x.y.A;
2
3  public class B {
4  }
5
6  package k.l;
7
8  import static x.y.A.B.*;
9
10 public class C {
11 }
```

Listing 4: Alternative example for an import statement in which a top-level class is imported.

Type-imports-on-demand can denote a type or package [9]. Based on the aforementioned strategy **S1**, a package is always assumed, and no element is created.

In case of a static-import-on-demand, the last part of the qualified name represents the imported member of a type whose name is the second-last part of the import. The simple recovery mechanism creates a class for the type and a static method for the imported member. Both elements receive the corresponding name. Additionally, the return type of the method is set to `void`, and its statement to an empty block. The previously handling of a method's creation will be applied on all methods in the simple recovery mechanism and will be referred to as the strategy **S2**.

Annotations The annotation interface of an annotation can come from either an import statement, the same package as the class which employs the annotation, or a type referenced by multiple name parts. In all three cases, the annotation interface with the name of the annotation is created. For methods of annotation values, strategy **S2** is applied.

References to Packages or Modules For referenced packages or modules, a corresponding root model element is created and added to the `JavaResource`. The name of the package or module is split into its individual parts which are set as the namespaces of the generated model element.

References to Types For every directly referenced type, e. g., a parameter type or the type of a local variable, a new class with the name of the type is created applying **S1**.

References in Expressions For constructor and method calls, strategy **S2** is executed except that a constructor has no return type and **S1** is applied on constructor calls if they contain multiple name parts.

For identifier references, it is difficult to determine the type of the actual referenced element. Therefore, for every identifier reference, a field with the name of the identifier is generated and added to the container class. The type of the field is set to `int`.

7.5.2 An Advanced Recovery Mechanism

The proposed advanced recovery mechanism tries to re-create the original source code as close as possible so that the printed models could be compiled. However, in certain cases, only a best guess is possible because the context does not provide appropriate information to properly reconstruct the original source code as outlined in the explanation for strategy **S1**.

In general, if a type with a known full qualified name has no model representation, it is created so that the resulting full qualified name of the model element equals the type's name requiring a new `CompilationUnit` for every type. New types will also be created. Usually, the actual types are unknown and receive an arbitrary valid type name, and a class is created for types if not explicitly stated otherwise. If a method or field is generated, it needs to be attached to a suitable type while existing methods and fields in the target type are checked as alternative elements before the generation because, e. g., method overloading is allowed, but methods with the same name cannot have different return types.

Import Statements The advanced recovery mechanism handles import statements similar to the simple recovery mechanism with some exceptions. The concrete created elements depend on their further usage in the model. If, e. g., an imported type is used as an annotation, an annotation interface is required. In contrast, if an imported type is part of the `implements` declaration of a class, it is an interface. For static-imports-on-demand, a method is generated if a method call has the name of the imported member. Otherwise, a field is created.

Annotations As in the simple recovery mechanism, an annotation can reference either an imported type, an annotation interface within the same package as the type employing the annotation interface, or a type with multiple name parts. In case of an imported type, the annotation can be resolved when the import statement is resolved. If the type is not imported and the annotation has a simple name, the annotation interface is located in the same package as the utilizing type so that the full qualified name of the annotation interface is the package name of the utilizing type combined with the annotation name and a corresponding type can be created. Otherwise, the annotation has multiple name parts where two cases need to be differentiated. If at least the first name part references an imported type, all following names refer to inner types which are created accordingly. Additionally, if a name includes type arguments, a type parameter is generated for every type argument and attached to the corresponding type. No bound is set for the type parameter while its name can be arbitrary, but must be valid. If the first name part of an annotation does not reference an imported type, the name is a full qualified name for which an annotation interface is created. If there is a name part with type arguments, it denotes a type so that a corresponding model element is generated. All name parts which follow result in inner types.

For every annotation value, a new method is created and added to the annotation interface. An annotation value is connected with a name which is used for the method. If there is no connected name, the default name `value` is set. Every generated method has no parameter and receives a return type which is inferred from the annotation value. At last, it declares no default value and has an empty statement.

References to Packages and Modules The advanced recovery mechanism handles references to packages and modules as the simple recovery mechanism.

References to Types Similar to annotations, references to types are handled. Instead of generating annotation interfaces, classes or interfaces are created. An interface is only selected for a type if the type reference occurs in a location in which only interfaces are allowed, e. g., in an `implements` declaration or as further types in cast expressions or type bounds of type parameters.

References in Expressions: Identifier Reference Conversion For every identifier reference, the advanced recovery mechanism usually considers the type of the previous reference at first. However, if the identifier reference is the first reference in a chain, two cases are differentiated as in annotations or type references. Thus, if the identifier reference is resolved, its type can be obtained. Else, a new class is created with the identifier as name located in the same package as the class containing the identifier reference.

When the previous type of an identifier reference is obtained, a new field is generated. The name equals the identifier of the identifier reference. If the previous reference is a direct type reference, the new field is static while its type is a new class. Otherwise, the new field is not static. Its type depends on the resolution of the previous reference. If the previous reference was recovered, the type of the new field is set to the type of the previous reference. Else, a new class is created and set as the type of the field. In specific contexts, and if an identifier reference is the last reference in a chain, the type for the field can be inferred from the context. A simple example is the assignment expression `int f = d.e;` where `e` needs to have the type `int`.

References in Expressions: Method Call to Method Conversion For a method call, a new method is generated. It receives the name used in the method call, and every argument is converted to a parameter. Thus, the number of parameters equals the number of arguments. While the parameter names can be arbitrary or follow a specific pattern, e. g., *param* followed by an increasing number, the parameter type equals the argument type if it can be inferred from the argument. If this is not possible because the argument contains nested references with unresolvable proxy objects, lambda expressions, or method reference expressions, the advanced recovery mechanism recursively applies to the argument resulting in a new or known type for the argument and parameter. In addition, if the method call declares type arguments, they are converted to type parameters as for annotations. Surrounding try-catch blocks of a method call are considered for exceptions thrown by the new method. For the return type of the new method, a new type is created if the context does not dictate the type similar to identifier references. Also similar to identifier references, the new method is static if the previous reference is a direct type reference and not static else. To conclude the conversion, if the new method is part of an interface, its statement is an empty statement. If it is part of a class, the statement is set to a block with a return statement returning `null`.

References in Expressions: Constructor Call to Constructor Conversion Analogous to method calls, a constructor call is converted to a constructor. Nevertheless, a constructor call references a type which is handled as the types in type references or annotations to get the concrete type element to which the new constructor is added. Compared to method calls, a constructor does not require a return type and can have type arguments for the referenced type and for the constructor. As a result, type arguments for the constructor are converted to type parameters of the constructor.

Lambda Expression to Type Conversion The target type of a lambda expression is a functional interface, i. e., an interface with exactly one non-static abstract method [9]. Therefore, a new interface is created for a lambda expression. To this interface, a new method with an arbitrary, but

valid name and empty statement is added. Every parameter of the lambda expression is converted to a parameter of the method. If the lambda parameters are explicitly typed without `var` as type, the method parameter type equals the lambda parameter type. Otherwise, every parameter receives a new type. In the body of the lambda expression, the advanced recovery mechanism is also applied because the body is considered for the return type of the new method. If it is a block without a return statement or with an empty return statement, the return type is `void`. In all other cases, the return type is inferred from the lambda body, i. e., it is the type of the expression if the lambda body is an expression, or the type of the return statements in a block if the lambda body is a block.

Method Reference Expression to Type Conversion There are two cases for method reference expressions if their type needs to be determined: either the referenced method cannot be resolved or they reference potentially multiple methods. In the latter case, one of the methods can be arbitrarily selected. For the first case, a new method is created. Because a method reference expressions also refers to a type, the new method will be added to this type. The name of the method equals the name of the referenced method while the method receives zero parameters, a block with a return statement returning `null`, and a new type as the return type.

With the aforementioned mechanism to obtain the method referenced by a method reference expression, the method can be converted to a method in a functional interface because the target type of a method reference expression is a functional interface [9]. Therefore, a new interface with a new method is generated. All attributes of the referenced method can be copied to the method of the functional interface.

8 Conclusion

This documentation presented the current state of JaMoPP. Compared to the original JaMoPP version, the meta-model was extended to support the features of Java 7-15. Furthermore, the dependency to EMFText was removed by implementing a manual printer and parser which is based on the Eclipse JDT Core. Three variants are available for the resolution of references between Java models. While the first variant generates complete models by directly setting the references, the second and third variant introduce proxy objects for references which can be resolved at a later point in time. In addition, the third variant offers a binding-based resolution which performs the resolution after the parsing and utilizes the bindings of the Eclipse JDT Core ASTs. In a comparison of all three variants, the average execution time indicate that the first variant is the fastest one while the proxy objects can support certain use cases with frequent accesses within the source code models and saving space. At last, topics for the potential future development of JaMoPP has been described. This includes the features of Java 16 and 17, the adoption of parts of Xtext to possibly reduce the maintenance effort, and recovery mechanisms to create model elements for unresolvable proxy objects.

References

[1] *Class ASTParser*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[2] *Class Type*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[3] *Class VariableDeclarationFragment*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[4] James Gosling et al. *The Java Language Specification, Java SE 10 Edition*. Feb. 20, 2018. URL: https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf.

[5] James Gosling et al. *The Java Language Specification, Java SE 11 Edition*. Aug. 21, 2018. URL: https://docs.oracle.com/javase/specs/jls/se11/jls11.pdf.

[6] James Gosling et al. *The Java Language Specification, Java SE 12 Edition*. Feb. 8, 2019. URL: https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf.

[7] James Gosling et al. *The Java Language Specification, Java SE 13 Edition*. Aug. 21, 2019. URL: https://docs.oracle.com/javase/specs/jls/se13/jls13.pdf.

[8] James Gosling et al. *The Java Language Specification, Java SE 14 Edition*. Feb. 20, 2020. URL: https://docs.oracle.com/javase/specs/jls/se14/jls14.pdf.

[9] James Gosling et al. *The Java Language Specification, Java SE 15 Edition*. Aug. 10, 2020. URL: https://docs.oracle.com/javase/specs/jls/se15/jls15.pdf.

[10] James Gosling et al. *The Java Language Specification, Java SE 16 Edition*. Feb. 12, 2021. URL: https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf.

[11] James Gosling et al. *The Java Language Specification, Java SE 17 Edition*. Aug. 9, 2021. URL: https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf.

[12] James Gosling et al. *The Java Language Specification, Java SE 7 Edition*. Feb. 28, 2013. URL: https://docs.oracle.com/javase/specs/jls/se7/jls7.pdf.

[13] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. Feb. 13, 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[14] James Gosling et al. *The Java Language Specification, Java SE 9 Edition*. Aug. 7, 2017. URL: https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf.

[15] James Gosling et al. *The Java Language Specification, Third Edition*. Addison-Wesley, June 2005, p. 688. URL: https://docs.oracle.com/javase/specs/jls/se6/jls3.pdf.

[16] Florian Heidenreich et al. "Closing the Gap between Modelling and Java". In: *Software Language Engineering*. Ed. by Mark van den Brand, Dragan Gašević, and Jeff Gray. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 374–383. ISBN: 978-3-642-12107-4.

[17] Florian Heidenreich et al. "Derivation and Refinement of Textual Syntax for Models". In: *Model Driven Architecture - Foundations and Applications*. Ed. by Richard F. Paige, Alan Hartman, and Arend Rensink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 114–129. ISBN: 978-3-642-02674-4.

[18] Florian Heidenreich et al. *JaMoPP: The Java Model Parser and Printer*. Tech. rep. 2009.

[19] IBM. *Übersicht zu Eclipse Modeling Framework (EMF)*. June 16, 2005. URL: https://www.ibm.com/support/knowledgecenter/de/SSQ2R2_9.5.1/org.eclipse.emf.doc/references/overview/EMF.html.

[20] Oracle Inc. *Java Language and Virtual Machine Specifications*. 2020. URL: https://docs.oracle.com/javase/specs/.

[21] *Integration with EMF*. Accessed: 19.12.2021. URL: https://www.eclipse.org/Xtext/documentation/308_emf_integration.html.

[22] *Interface IBinding*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[23] *Interface IPackageBinding*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[24] *Interface ITypeBinding*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[25] *JaMoPP*. June 9, 2019. URL: https://github.com/DevBoost/JaMoPP.

[26] *JDK 17*. Accessed: 20.01.2022. Sept. 14, 2021. URL: https://openjdk.java.net/projects/jdk/17/.

[27] Jóakim von Kistowski et al. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS '18. Milwaukee, WI, USA, Sept. 2018.

[28] *LANGUAGE ENGINEERING FOR EVERYONE!* Accessed: 19.12.2021. URL: https://www.eclipse.org/Xtext/index.html.

[29] *Language Implementation*. Accessed: 19.12.2021. URL: https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html.

[30] Object Management Group, Inc. *XML Metadata Interchange (XMI) Specification - Version 2.5.1*. June 7, 2015. URL: https://www.omg.org/spec/XMI/2.5.1/.

[31] *Package org.eclipse.jdt.core.dom*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.