Recovering Missing Dependencies in Java Models

Martin Armbruster
martin.armbruster@kit.edu

Manar Mazkatli
manar.mazkatli@kit.edu

Anne Koziolek
koziolek@kit.edu

Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany

Abstract

Different approaches use models of source code to extract performance models from the code which allow performance predictions and the exploration of design alternatives. The extended Java Model Parser and Printer provides a modeling environment for Java code. It defines a metamodel and contains a parser and printer including three variants to resolve references between different Java models. These variants assume that the complete code with all dependencies is available or missing elements are not accessed.

In this paper, a trivial recovery strategy is introduced. It is able to recover references which cannot be resolved. Additionally, the performance and model storage of the reference resolution's variants are compared with and without the trivial recovery. The results indicate that the trivial recovery reduces the execution time and required space for storing the models. In the future, further recovery strategies can be implemented to allow a balance between performance and model accuracy.

1 Introduction

Architecture-level performance models allow performance predictions to, for example, explore design alternatives. There are different approaches to extract such performance models from source code using models of the code as input (e.g., the Continuous Integration of architectural Performance Models approach [7]). Thus, they need a component which at least defines a metamodel for the targeted programming language and is able to parse the source code to create a model. Depending on the use case, these models should be valid (i.e., conforming to the metamodel) and accurate (i.e., they represent the original code one-to-one). As code has external dependencies, the dependencies are also part of the code models and affect the validity and accuracy. However, they can be considered less relevant compared to the actual source code if the focus lies only on the code as in [7]. Moreover, in [7], source code is continuously parsed so that it requires a fast parsing.

The Java Model Parser and Printer (JaMoPP) and the extended JaMoPP enable the modeling of Java code (as outlined in Section 2). Currently, the extended JaMoPP assumes that the full source code and all dependencies are available. Otherwise, the models are incomplete and invalid. Therefore, this paper contributes a trivial recovery strategy (in Section 3) to obtain valid and less accurate models by creating model elements for missing dependencies. Afterwards (in Section 4), its performance compared to no recovery is investigated. These results indicate an improvement in the performance. Finally, the paper concludes with future work (in Section 5) in which advanced strategies will be explored to find a balance between performance and accuracy.

2 Foundations

This section introduces JaMoPP and its extension.

2.1 The Java Model Parser and Printer

JaMoPP provides a modeling environment for Java source code based on the Eclipse Modeling Framework (EMF) [1]. Thus, it contains a metamodel which supports the syntax of Java 6 [6]) and an automatically generated parser for a code-to-model conversion and printer for a model-to-code conversion [1].

To connect the references between different Java models introduced by, e.g., imports, JaMoPP contains a Java-specific reference resolution [1]. During the parsing, a temporary proxy object is created for every reference which is resolved when it is accessed (i.e., the proxy object is replaced with the actual and correct model element) [2, 5]. If the reference points to an element for which the corresponding file was not loaded before, JaMoPP parses the file on demand [1].

2.2 The extended JaMoPP

Because JaMoPP is limited to Java 6 and newer Java versions are available, the extended JaMoPP[1] is independently developed to enhance JaMoPP with the features of new Java versions [6]. Currently, it supports the language features of Java 7-15 including lambda expressions, modules, and switch expressions. These new features in the metamodel are complemented by a new printer and parser implementation.

In the case of the parser, the extended JaMoPP utilizes the Eclipse Java Development Tools (JDT) Core to parse Java code into abstract syntax trees (AST) and to convert the ASTs into Java models.

[1] https://github.com/MDSD-Tools/TheExtendedJavaModelParserAndPrinter

Similar to the original JaMoPP version, the extended JaMoPP provides three variants for the reference resolution [6]. In the first variant, all references are directly resolved and set during parsing without generating proxy objects. Thus, it relies on the JDT bindings which provide connections between different AST elements[2]. In contrast, the second variant builds upon the original JaMoPP's reference resolution extending it to support the new Java features [6]. At last, the third variant combines the first and second one by still creating proxy objects for references. However, during parsing, the proxy objects are partly resolved with the help of the bindings and depending on the parser options. After the parsing, remaining proxy objects are resolved with the second variant.

3 A Recovery Strategy

A comparison of the reference resolution's three variants reveals that they have different assumptions about the code and its dependencies [6]. While the first variant assumes that the complete code and all dependencies are available, the second and third variant do not require such a completeness when references to missing code or dependencies are not accessed. As a consequence, if the assumption for the first variant does not hold or a reference to missing code or dependencies is accessed (i.e., the corresponding proxy object cannot be resolved), the resulting models are incomplete and invalid.

In order to obtain valid models focusing on the second and third variant as initial step, a trivial recovery strategy (introduced in version 6.0.0 of extended JaMoPP) is able to replace proxy objects with actual model elements. The strategy takes a proxy object, creates a new model element with the same type, enriches the new element with required attributes, and replaces the proxy object with the new element. In every step, no context information is considered.

The code example in Listing 1 consists of a class `B` as part of the dependencies and class `A` which represents a part of the source code and calls the method `c` in class `B`. During parsing, the dependency with class `B` is not available so that the method call to `c` cannot be resolved. Figure 1 shows a simplified model of class `A` before and after executing the recovery. Before, it has a proxy object for the call to `c`. Afterwards, the proxy object is replaced with a non-proxy model element. Additionally, the `typeReference` of the created method element representing the return type of the method is set to `void` because the context of the method call is ignored. As every method must be contained within a class, the trivial recovery creates the artificial class `SyntheticClass` which acts as a container for every recovered method and field and adds the created element for `c` to this artificial class.

Although the trivial recovery allows to obtain valid

[2]See `https://github.com/eclipse-jdt/eclipse.jdt.core/tree/de66fd3d3a56b95e1a62cc578aeb46a6b45a254d`

```
1  // Part of source code.
2  class A {
3       void d() {
4            B b = new B();
5            int i = b.c() + 5; } }
6  // Part of dependency.
7  class B {
8       int c() {} }
```

Listing 1: Code example with source code and dependency code.

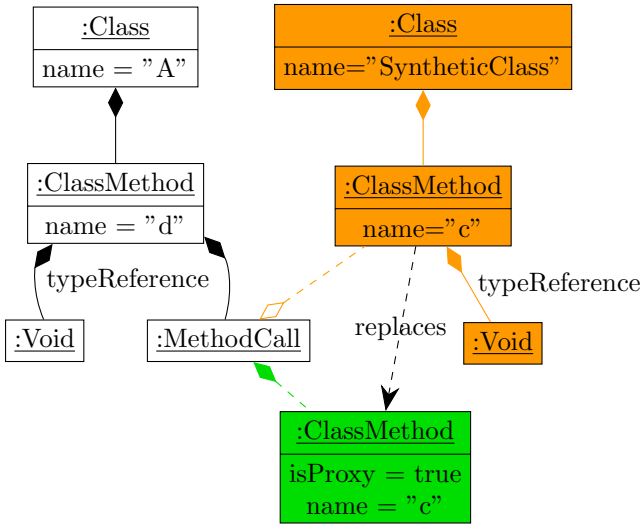

Figure 1: Simplified Java model for the code example. The orange elements replace the green ones during the recovery.

models, they can be inaccurate. Looking back at Listing 1, the recovered method element for `c` is contained in an artificial class instead of the class element for `B` which is also recovered for the type in the variable declaration of `b`. Therefore, advanced recovery strategies are proposed that aim to provide more accurate models by analyzing the context of the proxy objects [6]. In the example, for recovering the method `c`, they would consider the type of the variable `b` and would add the recovered method element to this type.

4 Performance Evaluation

Aside from the goal to provide valid models, another goal of the trivial recovery strategy is an improvement in performance and model storage as previous initial measurements indicate high execution times without a recovery [6]. As a result, this evaluation tries to answer the following question: how does the recovery improve the performance of the reference resolution and model storage compared to no recovery?

Because the recovery currently targets the second and third variant, the first variant is not considered. Regarding the second and third variant, three configurations are selected: the plain second variant (SV),

	With Recovery	Without Recovery
SV	78.1 s (std. 4.4 s)	> 1 hour
OL	198.1 s (std. 10.7 s)	> 1 hour
FR	does not apply	216.2 s (std. 12.3 s)

Table 1: Average execution times with standard deviation and different configurations.

		Source Code Only / Total
SV with recovery	#Files	160 / 934
	Size	18.0 MiB / 305.6 MiB
OL with recovery	#Files	160 / 1203
	Size	13.8 MiB / 19.7 MiB
FR	#Files	160 / 3957
	Size	13.8 MiB / 101.1 MiB

Table 2: Results of calculating the required storage to save models from different configurations.

the third variant in which only the bindings of the source code models are resolved (leading to the resolution of direct references into the dependencies called one level configuration, *OL*), and the third variant in which all occurring bindings and remaining proxy objects are resolved as part of the parsing process (full resolution, *FR*).

In every configuration, the parser gets the full source code of the TeaStore in version 1.4.0 as input. TeaStore is a web-based store for tea and related products designed for tests and benchmarks [4]. During the evaluation, the following execution times are measured 100 times with a timeout of 1 hour to calculate the average and standard deviation[3]: the total parsing time including resolutions and the recovery if it is executed. In the case of FR, no recovery needs to be performed. For SV and OL with recovery, the proxy objects within the source code models are resolved. Afterwards, the trivial recovery is executed.

To measure the model storage, all models of each configuration are output in the XML Metadata Interchange (XMI) format [3], and the total number of files and their size as well as the number of source code models and their size are counted.

Table 1 displays the execution times for all configurations. OL and SV without recovery exceed the timeout. While FR requires 216.2 s, it is outperformed by OL and SV where SV in turn outperforms OL. As shown in Table 2, all configurations provide the same number of models for the source code (160). SV requires 4.2 MiB more storage for these models. In total, SV generates the smallest file number with 934 files while FR creates 3957 files. However, SV requires the most storage with 305.6 MiB. In contrast, OL only requires 19.7 MiB.

As expected, the recovery speeds up the process in SV and OL to obtain valid models. It is faster than FR, too. Nevertheless, the models of FR are valid and accurate leading to the difference in the number of files and storage between OL and FR. OL contains only a subset of the accurate models of FR. Remaining references point to recovered elements decreasing the required storage. When SV resolves the proxy objects in the source code, it loads additional files including statements which are not present in the JDT bindings. As a result, it increases the model size. At the same time, it seems that this additional file loading reduces the number of total files compared to OL and FR. All in all, the trivial recovery can improve the performance and model storage.

[3]Computer: Intel Core i5-7200U, 8GB RAM, Windows 10.

5 Conclusion and Future Work

In this paper, the trivial recovery strategy for the extended JaMoPP was introduced. It provides model elements for proxy objects in the reference resolution's second and third variant by creating new valid elements with the same type as the proxy object. As a consequence, users can obtain valid models with improved performance and model storage. In future work, the trivial recovery can be compared to proposed advanced recovery strategies [6] potentially offering a balance between performance, model storage, and model accuracy. Furthermore, the proposed recovery strategies can be applied to the extraction of architecture-level performance models and different code sizes in order to investigate the generalizability, applicability, and scalability.

References

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

[2] *emftext USER GUIDE*. Accessed: 26.07.2023. Sept. 27, 2012.

[3] Object Management Group, Inc. *XML Metadata Interchange (XMI) Specification - Version 2.5.1*. June 7, 2015.

[4] J. 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.

[5] IBM Corporation. *Understanding Models*. Accessed: 26.07.2023. Mar. 5, 2021.

[6] M. Armbruster. *Parsing and Printing Java 7-15 by Extending an Existing Metamodel*. Tech. rep. July 28, 2022.

[7] M. Mazkatli et al. "Continuous Integration of Architectural Performance Models with Parametric Dependencies – The CIPM Approach". In: (2022).