Models from Code or Code as a Model?

Antonio Garcia-Dominguez[1] and Dimitrios S. Kolovos[2]

[1]School of Engineering and Applied Science, Aston University, UK
a.garcia-dominguez@aston.ac.uk

[2]Department of Computer Science, University of York, UK
dimitris.kolovos@york.ac.uk

Abstract. Many model-driven engineering workflows take the existing code of a system as an input. Some of these include validation and verification, software modernisation or knowledge extraction. Current approaches take the code and produce a standalone model, instead of treating the code itself as a model and navigating through it on demand. For very large codebases this can be quite expensive to produce, and the resulting model can be too large to suit the simplest monolithic file-based stores. In this work we propose reusing a model that is already incrementally built for us and supports fast lookups: the internal indices maintained by our integrated development environments for code analysis and refactoring. In particular, we show an Epsilon driver that exposes the Eclipse JDT indices of one or more Java projects as a model, and compare its performance and usability against MoDisco for validating Java code with regards to a partial UML model.

1 Introduction

The ability to query codebases (e.g. written in languages like Java and C#) at the abstract syntax level – as opposed to the concrete syntax level which is supported by standard textual search facilities – is essential for tasks such as program comprehension, analysis, quality assessment, reverse engineering and modernisation. To query such a codebase using contemporary EMF-based model management languages (e.g. ATL or Acceleo), it first needs to be transformed into an EMF-compatible representation using tools like MoDisco [1]. This practice presents a number of issues. First, as none of the existing code-to-model tools are incremental, after any changes to the codebase, the latter needs to be re-transformed into an EMF representation, which can be a rather time consuming task as the code grows in size. Moreover, for queries that do not need to navigate the complete abstract syntax tree, such a transformation can be wasteful.

To address these issues in this paper we present a more lightweight and performant approach for accessing Java codebases using the model management languages of the Epsilon platform. In the proposed approach, instead of transforming the codebase into an EMF-based representation, we provide an adapter for the facilities of the Eclipse Java Development Tools (JDT). The adapter exposes the internal representation of the code maintained by JDT as a set of

"models" that, as far as Epsilon is concerned, are indistinguishable from any other type of model. We also demonstrate how we use advanced JDT features such as its search facilities to support performant OCL-inspired *select* queries.

The rest of the paper is structured as follows. Section 2 presents some key events in the history of model-driven reverse engineering, and summarises the current state of the art in terms of specifications and tools. Section 3 describes the proposed approach. Section 4 compares the performance of MoDisco against two configurations of our approach through a case study in which a codebase is validated against a partial UML model. Finally, Section 5 presents our conclusions and lines of future work.

2 Background: models from code

The practice of reverse engineering (RE) has been around for a long time, especially in the field of hardware design, where it is common to study finished products to extract high-level designs. Chikofsky and Cross [2] provided a taxonomy of earlier software RE efforts, mentioning the ability to extract documentation and design artifacts, to restructure the code, or to reengineer it (produce a new version from a higher-level model extracted from the code).

In 1998, Kazman et al. presented CORUM II [3], an reengineering tool integration model that consisted of an initial bottom-up "architecture recovery" process (from code to higher-level representations) followed by a top-down "architecture-based development" (from the transformed higher-level representations to the new code). This bottom-up and top-down combination is commonly known nowadays as the "horseshoe" approach for reengineering, and is still being used in modern tools, as mentioned below.

The term "model-driven reverse engineering" (MDRE) term was coined more recently: one of its first uses was by Rugaber and Stirewalt in 2004, to present their work on extracting and enhancing a model of a root solving program and generating an equivalent program [4]. Rugaber and Stirewalt argued that the formality of the specifications and the support for automated validation and code generation simplified the production of the new version of the system and reduced the risk of mistakes.

This need for formality and standardisation was also considered by the Object Management Group (OMG), which in June 2003 issued a whitepaper inviting contributions to the Architecture-Driven Modernization Task Force (ADMTF) [5]. The efforts of the ADMTF produced version 1.0 of the ADM Knowledge Discovery Metamodel (KDM) in January 2008: the latest formal release of KDM (version 1.3) is from August 2011 [6]. KDM is a comprehensive metamodel that can express the various types of knowledge extracted from a legacy system, covering not only the structure of the code but also UI, data modelling, event handling and other concerns.

KDM was designed to cover the high-level concepts present in a legacy system, rather than provide a 1:1 representation of the original sources. To cover that gap, the ADMTF published the Abstract Syntax Tree Metamodel (ASTM)

1.0 specification in January 2011 [7]. ASTM is a high-fidelity, low abstraction level metamodel that is divided into a generic AST metamodel (GASTM) and a set of language-specific AST metamodels (SASTMs).

There have been various implementations of the KDM and ASTM metamodels: MoDisco in particular is considered by the OMG as the reference implementation of the KDM and ASTM specifications [1]. MoDisco is an open source project, part of the Eclipse Modeling top-level project. It is based on the Eclipse Modeling Framework (EMF), and includes the various types of components required to enable MDRE (discoverers, transformations and generators). MoDisco is largely divided into a base "infrastructure" layer (management of discoverers, KDM/ASTM metamodels, etc.), a middle "technologies" layer (specific language support, e.g. Java programs or Java Servlet Pages scripts), and a top-level "use cases" layer (workflows of MoDisco invocations). MoDisco uses language-specific metamodels for the discoverers, whose models can be then transformed to the "pivot" metamodels (KDM and ASTM) for interoperability. In the same paper, Bruneliere et al. show how MoDisco was used by Mia-Software for refactoring and quality metric monitoring. The general approach of MoDisco is similar to what was proposed in CORUM II, but its design allows for more flexibility in terms of internal representations and support for other languages.

Other MDRE tools include:

- JaMoPP[1] can extract EMF-compatible models conforming to a custom Java metamodel from Java source code, using a Java parser generated through EMFText [8].
- Moose[2] is a Smalltalk-based tool that provides a platform for customised source code analyses, based on models conforming to their FAMIX metamodel [9]. FAMIX is a generic metamodel dedicated to representing the underlying constructs of any object-oriented programming language.
- Rascal[3] is a metaprogramming environment that allows for scripting model transformations, and it includes a library that extracts a Rascal-compatible model from the Eclipse JDT representations of a Java project [10].
  Rascal's JDT library is the closest to our approach, but it operates on an extracted model instead of working directly with the JDT representations, and it does not give access to the full AST. Another limitation is that Rascal currently does not support EMF-based models, unlike Epsilon: these are useful for case studies such as the one showed in Section 4.

3 Epsilon JDT driver: code as models

The previous section discussed various approaches for extracting models from code, producing standalone representations that can be managed with EMF-compliant tools. However, these extractors are usually one-off processes: if the

[1] `http://www.jamopp.org/index.php/JaMoPP`
[2] `http://www.moosetechnology.org/`
[3] `http://www.rascal-mpl.org/`

code changes, the extractor needs to be run again from scratch. Implementing an extractor that takes advantage of the modularity of the target programming language to achieve incrementality is not a simple task, and state-of-the-art model extraction tools do not support this.

On the bright side, there is already a class of software that implements such incremental extractors: integrated development environments (IDEs). Modern IDEs provide advanced code search and refactoring tools, and in order to provide a responsive experience they need to build, maintain and use their internal representations and indices quickly and efficiently. This paper proposes exposing these representations as models, instead of writing one-off batch extractors.

This section describes a first implementation of such an approach, in which the representations and indices of the Eclipse Java Development Tools are exposed as models for the Eclipse Epsilon [11] family of model management languages. An overall description of the design is given, followed by a more specific discussion about the integration with the search capabilities of JDT.

3.1 Overall design and implementation

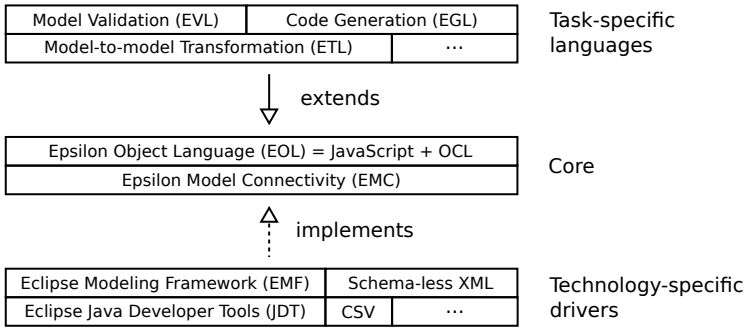

Fig. 1. Architecture of the Epsilon family of model management languages

Figure 1 shows the general architecture of the Epsilon family of model management languages. The Epsilon Object Language (EOL) serves as the base language for the rest of the Epsilon languages, which are specialized for certain tasks (e.g. EVL for validation). Reading and writing models is abstracted over the Epsilon Model Connectivity (EMC) layer, which supports models implemented in the Eclipse Modeling Framework (EMF), plain XML files, CSV files and so forth. This work presents a first version of a new "technology-specific driver" which exposes the representations maintained by the Eclipse Java Developer Tools (JDT) as models: the *EMC JDT model driver*. The latest version of the model driver is available as open-source software on GitHub[4].

[4] `https://github.com/epsilonlabs/emc-jdt`

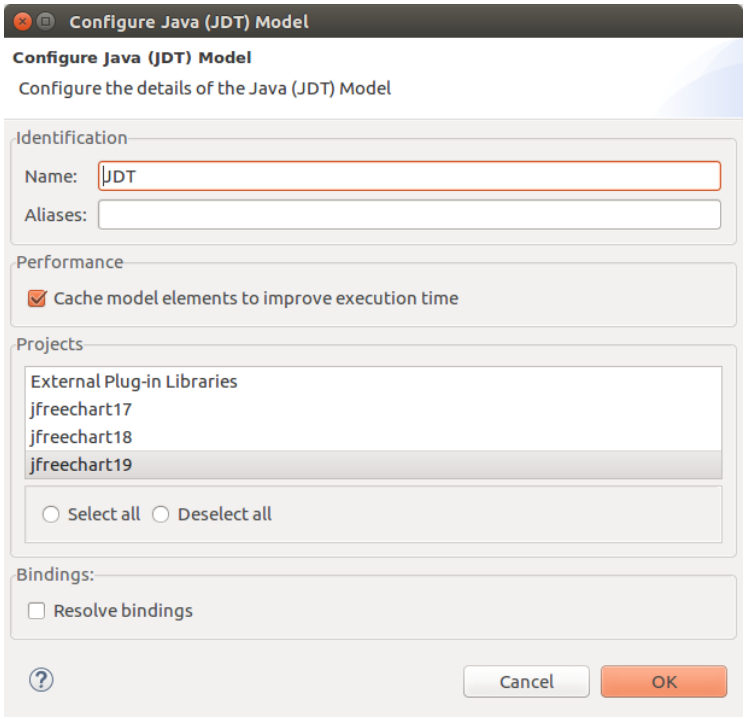

Fig. 2. Screenshot of the configuration dialog for an EMC JDT model

The entry point of an Epsilon model driver is its implementation of the Epsilon *IModel* interface: for the EMC JDT driver, it is the *JdtModel* class. The dialog used to configure an EMC JDT model is shown in Figure 2: the model takes in the names of the Eclipse Java projects that should be exposed as a model and whether Java bindings across files (i.e. resolutions of local names to external types, functions and fields) should be precomputed. These bindings may produce useful information needed for the model management task at hand, but are generally expensive to compute.

Upon load, *JdtModel* fetches the actual projects and prepares a *ReflectiveASTVisitor* that only traverses the Java source files in those projects. *ReflectiveASTVisitor* is a specialisation of a JDT *ASTVisitor*, an implementation of the Visitor design pattern [12]. When the user issues an EOL query of the form `X.allInstances`, *ReflectiveASTVisitor* parses all Java source files in those projects, traverses their JDT Document Object Model (DOM) abstract syntax trees and returns all the instances found of the JDT DOM *X* class. This filtering is done through Java reflection, by checking the names of the class and superclasses of each object. This list of instances can be cached in memory (through a respective configuration flag) to reduce times for future queries.

The EMC JDT driver also provides some convenient shorthands to access commonly required information, through an implementation of an EMC *IPropertyGetter*. For a JDT *FieldDeclaration* `f`, its name can be accessed through `f.name`: since the underlying JDT DOM class does not have a `name` property, name clashes are avoided. *BodyDeclaration*s are extended as well with simple `public`, `static`, `abstract`, `protected`, `private` and `final` Boolean fields, instead of having to loop through *Modifier* instances.

3.2 Integrating Eclipse indexing with EOL

So far, we have discussed how the EMC JDT driver uses the Java parsing facilities of JDT in order to iterate throughout a codebase and find all instances of a particular language construct. However, in many scenarios, what is needed is a quick way to retrieve specific instances instead (e.g. a Java type declaration with a specific name). This is well supported by the code navigation and refactoring capabilities of JDT, which keeps incrementally indices of all the Java projects in the running Eclipse IDE. These are essentially memory- and disk-backed hash tables from category/key pairs (e.g. "class definition" and class name) to the path of the relevant `.jar` or `.java` file. Indices are reasonably lightweight: in a workspace with the Epsilon source code (400k+ SLOC of Java code), the largest `.index` file was 11MB.

As a proof of concept, the EMC JDT driver was extended to use the JDT indices to quickly look up JDT *TypeDeclaration*s by name. `TypeDeclaration .allInstances` was specialized to postpone the traversal of the source files and immediately return a custom *SearchableTypeCollection* (STC), which can optimise specific queries on the collection through the Eclipse indices and otherwise fall back to regular traversal. For an STC `c`, these queries are:

- `c.select(it|it.name = expr)`: the STC takes the string result of evaluating the included `expr` EOL expression and uses the JDT *SearchEngine* class to issue a query to quickly locate the JDT *ICompilationUnit* containing the desired type declaration. This compilation unit is parsed into a DOM and the appropriate DOM node is retrieved. This is faster than iterating through all the source code, but the file I/O and parsing still takes time.
- `c.search(it|it.name = expr)`. It operates in much the same way as `select`, but instead of parsing the file again to produce a DOM node, it returns the *JDT SourceType*s directly as indexed by Eclipse. This is noticeably faster, as no parsing is required, but it imposes more complexity on the user as they have to deal with one more *JDT* class.

These two queries illustrate an interesting point in the design of JDT: the *SearchEngine* does not return DOM nodes as one would reasonably expect. Instead, it returns internal representations that are only sometimes mapped back to DOM nodes, for the sake of performance. Users need to decide whether simplicity or performance is more important for their particular situation.

Another point is that *SearchEngine* has considerably more functionality, with the ability to search other types of elements (e.g. method and field declarations)

and provide various methods for filtering results. Integrating all these options into the STCs would be an interesting line of future work.

4 Case study: validating code against UML models

In the previous sections, two approaches have been discussed: one used standalone extractors to produce models from code, and another reused the incrementally built representations and indices of the Eclipse IDE as a model. This section compares both approaches for a specific case study: checking if a Java codebase implements all the classes and methods mentioned in a UML model. This case study is meant to represent a typical model-driven task that combines standalone models with a changing codebase.

The rest of this section will introduce the specific models and codebase used and the implementations of the validation task. After that, their performance results will be shown, and a discussion of their relative merits will follow.

4.1 Input models and codebases

For the experiment, it would have been ideal to have access to an large-scale industrial open-source system that had a corresponding and reasonably up-to-date and detailed UML diagram. However, these are hard to come by. As an approximation, the experiment will use the UML models available through `opensource-uml.org`, which were extracted from the Java code of several popular open source libraries using the Modelio[5] tool. In particular, the 41MB UML XMI file produced from JFreeChart 1.0.17 was used.

Next, the source code for versions 1.0.17, 1.0.18 and 1.0.19 of JFreeChart was downloaded from the official website[6]. Maven was used to generate Eclipse projects from the source releases, which were used unmodified save for some classpath settings to solve compilation issues. The Java discoverer in MoDisco 0.13.2 (running within Eclipse Mars.1, version 20150924-1200) was then used to extract Java models in XMI format from these codebases (one XMI file per version of JFreeChart). It must be noted that the Java discoverer also reuses parts of Eclipse JDT and implements its own *JDTVisitor* The extractions and all other experiments were run on a Lenovo Thinkpad T450 laptop with an i7-5600U CPU, 16GiB of RAM and a 256GB SSD running Ubuntu 16.04, Linux 4.4.0-28 and Oracle Java 8u60.

The MoDisco UI was instrumented to collect extraction times[7]: specifically, two lines of code were added to the `widgetSelected` method of the *MoDiscoMenuSelectionListener* to measure the wall time taken by the extraction. The extraction was run 10 times for each JFreeChart version: on average, extraction of the 1.0.17, 1.0.18 and 1.0.19 codebases required very similar times: 19.38s, 20.76s and 19.97s, respectively. This is to be expected, since all three codebases

[5] `http://www.modelio.org/`
[6] `http://www.jfree.org/jfreechart/`
[7] `https://gist.github.com/bluezio/830245299f11af5660c440668fc78d95`

are of similar size: the resulting XMI files took up 169MB, 173MB, and 174MB respectively. According to the JFreeChart forum[8], 1.0.18 added JavaFX support and some minor features, and 1.0.19 was a maintenance release.

4.2 Validation task

The validation task was implemented in the Epsilon Validation Language (EVL) [13], which specialises EOL into a rule-based notation for checking invariants across instances of certain types. Three implementations were written: one using the model extracted by MoDisco, and two using the EMC JDT model driver discussed in Section 3. One of the EMC JDT implementations uses `select`, and the other uses `search`: their relative merits were discussed in Section 3.2.

The three EVL scripts are very similar, and use conceptually the same rules. The EVL script for the `search`-based EMC JDT implementation is shown in Listing 1. Since the UML model includes many irrelevant classes (e.g. a reverse-engineered UML model of the JDK), the rules for UML classes are "lazy" and are only triggered from the `satisfiesAll` call of the rule for the UML packages whose fully qualified names start with the JFreeChart prefix.

The UML class rules check that: a) the Java codebase has exactly one matching type for each class (based on its name and the name of its containing class, if it exists), b) the matching type implements all the operations in the UML class as methods, and c) the nested classes obey the same rules. No validation errors are found for the 1.0.17 codebase (which is obvious, as the UML model was extracted from it), and the same 4 validation errors are found for 1.0.18 and 1.0.19, since some methods were removed from 1.0.17 to 1.0.18.

4.3 Performance results and discussion

Having prepared the models and the validation scripts, the final step was running the task itself. Each implementation was executed 10 times using the coarse profiling capabilities of an interim release of the Epsilon framework (commit 3d4408d) and the environment described in Section 4.1. Epsilon measured and reported the time required to load the model and perform the validation.

Figure 3 shows the times required for model extraction, model loading and the validation itself. The times for the various releases of JFreeChart were roughly equivalent. Focusing on the validation against the JFreeChart 1.0.17 source code, MoDisco took 36.25 s on average in total, EMC JDT with `select` took 23.16 s and EMC JDT with `search` took 12.30 s.

While these results are quite positive for EMC JDT, there are some considerations that need to be made:

- MoDisco had the lowest validation times of all three options: 3.20 s on average, compared to 5.09 s on average for EMC JDT with `search`. This is due to the fact that once the MoDisco model is loaded in memory, it is completely standalone and does not require performing any type of disk I/O.

[8] `http://www.jfree.org/forum/viewforum.php?f=3`

```
context UML!Package {
  guard : self.fqName().startsWith('jfreechart.org.jfree')
  constraint AllClassesValid {
    check {
      self.fqName().println('checking package ');
      return self.packagedElement.forAll(cl: UML!Class |
        cl.satisfiesAll('HasOneMatchingType', 'MethodsExist', 'NestedAreValid'));
    }
  }
}

context UML!Class {
  @lazy
  constraint HasOneMatchingType {
    check : self.matchingTypes().size = 1
  }
  @lazy
  constraint MethodsExist {
    check {
      var td = self.matchingTypes().first;
      return self.ownedOperations.forAll(op |
         td.methods.exists(tdMethod | tdMethod.name = op.name ));
    }
  }
  @lazy
  constraint NestedAreValid {
    check : self.nestedClassifier.forAll(n: UML!Class |
      n.satisfiesAll('HasOneMatchingType', 'MethodsExist', 'NestedAreValid'))
  }
}

@cached
operation UML!Class matchingTypes() : Sequence(JDT!TypeDeclaration) {
  var candidates = JDT!TypeDeclaration.all.select(td|td.name=self.name);
  if (self.eContainer.isKindOf(UML!Class)) {
    var parentClassName = self.eContainer.name;
    candidates = candidates.select(c|c.parent.isKindOf(JDT!TypeDeclaration)
      and c.parent.name = parentClassName);
  }
  return candidates;
}

@cached
operation UML!NamedElement fqName() : String {
  var container = self.eContainer;
  if (container.isDefined() and container.isKindOf(UML!NamedElement)) {
    return container.fqName() + '.' + self.name;
  } else {
    return self.name;
  }
}
```

Listing 1. EVL source code of the `search`-based EMC JDT implementation of the validation task. It checks that all UML classes and methods were implemented.

On the other hand, this relies on the fact that the extracted MoDisco model fit completely into memory. If it had been much larger, it would be necessary to use a database-backed store with support for lazy loading and unloading (e.g. CDO). This would change the performance profile of the task, potentially reducing model loading times (due to lazy loading) at the cost of validation times (due to additional disk I/O).

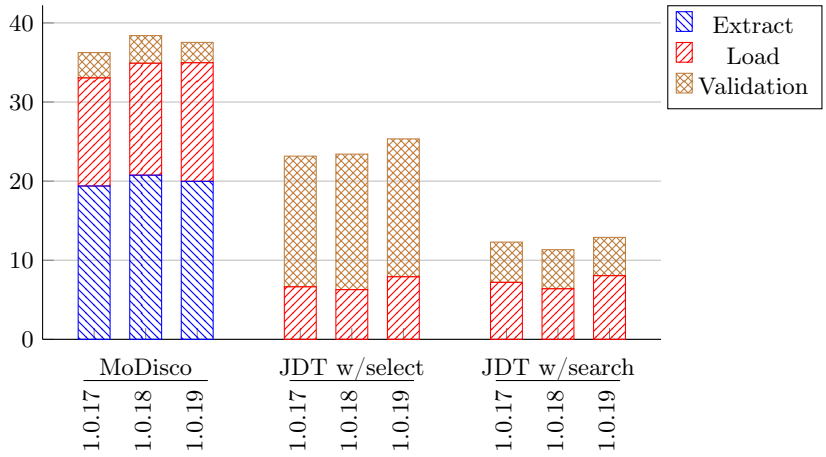

Fig. 3. Average processing times over 10 runs in seconds per implementation of the validation task, step of the workflow and version of the JFreeChart library.

- MoDisco had the highest model loading times, since it had to load a 169MB+ XMI file in addition to the 41MB UML model. In comparison, the EMC JDT models load almost instantly, as they retrieve model elements on demand: the model loading time for the JDT scenarios is dominated by the time required for the 41MB UML model.
- Outside this case study, the 20 seconds used for MoDisco model extraction could have been amortised over multiple queries. In the extreme case in which model extraction times were excluded altogether, the MoDisco approach would have taken 16.87 s on average for 1.0.17, faster than EMC JDT with `select` and only 4.57 s slower than EMC JDT with `search`.
  This suggests that EMC JDT would be most useful when working with frequently-changing codebases. Since every change would require extracting the MoDisco model from scratch, amortising its cost would be harder, and working "on demand" like EMC JDT would be more attractive.

Reiterating the third point, Figure 4 shows how much time would be required on average in total to validate the codebases of 1.0.17, 1.0.18 and 1.0.19 against the provided UML model. MoDisco would take 112.22 s, having to repeat the extraction for each release. JDT with `select` would take 71.92 s, avoiding the extraction but still having to reparse in order to map the search result to the DOM node. JDT with `search` would only take 36.52 s, avoiding the pitfalls of the two previous solutions.

5 Conclusion and future work

Current tools for querying a codebase as a model require an expensive model extraction step that needs to process all the source code (even if not all of it

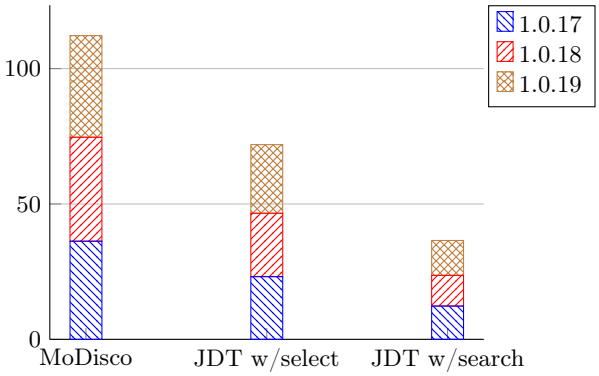

Fig. 4. Comparison of the total time required on average (in seconds) to process all JFreeChart versions, by implementation.

is needed), and must be repeated every time the code changes. In this work we have presented a more lightweight approach that adapts "on-demand" Eclipse JDT representations so they can be used from the languages in Epsilon as any other model. We have showed how to integrate the JDT indices transparently as well, making it possible to quickly find the relevant parts of the program for the query and reduce the amount of parsing needed.

The approach has been demonstrated with a case study in which multiple versions of a codebase were validated against a UML model: one of the configurations of the approach was 3x faster than MoDisco (36.52 s instead of 112.22 s). While the validation with MoDisco runs very quickly once the model is loaded into memory, we have found that the model extraction and model loading processes make it slower than our more lightweight approach. This suggests that MoDisco is mostly suited towards stable codebases (e.g. legacy systems), in which model extraction efforts can be amortised over many queries, and model loading can be solved by using a store that supports on-demand loading and custom indexing (e.g. Connected Data Objects). On the other hand, our solution is better suited than MoDisco for frequently changing codebases, in which the model extraction costs cannot be amortised and managing a database-backed representation only adds management overhead.

The present work has showed an initial prototype of the Epsilon JDT driver: we intend to perform some optimisations to reduce times even further, e.g. by caching parsed compilation units and integrating the other capabilities in the JDT search indices. Supporting the additional filtering capabilities (e.g. search by superclass) might require extending the OCL-like `select` operation in new ways. Further use cases may also reveal the need for additional property getters that encapsulate certain common queries over the JDT representations. Another avenue of work is reimplementing this lightweight "adapter" approach on other

languages supported by Eclipse, such as C or C++ (through the Eclipse CDT project[9]).

Acknowledgements

This research was part supported by the EPSRC, through the Large-Scale Complex IT Systems project (EP/F001096/1) and by the EU, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (grant #611125).

References

1. Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. MoDisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032, August 2014.
2. Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.
3. Rick Kazman, Steven G. Woods, and S. Jeromy Carrière. Requirements for integrating software architecture and reengineering models: CORUM II. In *Proceedings of 5th Working Conference on Reverse Engineering*, pages 154–163. IEEE, 1998.
4. Spencer Rugaber and Kurt Stirewalt. Model-driven reverse engineering. *IEEE software*, 21(4):45–53, 2004.
5. Object Management Group. Why do we need standards for the modernization of existing systems? Whitepaper, July 2003.
6. Object Management Group. Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM) 1.3. Formal specification, August 2011.
7. Object Management Group. Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM) 1.0. Formal specification, January 2011.
8. Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the gap between Modelling and Java. In *International Conference on Software Language Engineering*, pages 374–383. Springer, 2009.
9. Tudor Girba. The Moose book. `http://www.themoosebook.org/book/`, 2011.
10. Mark Hills, Paul Klint, and Jurgen J. Vinju. Scripting a Refactoring with Rascal and Eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools*, Rapperswil, Switzerland, June 2012. ACM New York, NY, USA.
11. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N. and Polack, F.A. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conf. on Engineering of Complex Computer Systems*, Potsdam, Germany, 2009.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison Wesley, Reading, Mass, first edition, October 1994. ISBN 978-0-201-63361-0.
13. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Rigorous methods for software construction and analysis. chapter On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages, pages 204–218. Springer-Verlag, Berlin, Heidelberg, 2009.

[9] `https://eclipse.org/cdt`