Working With Reverse Engineering Output for Benchmarking and Further Use

David Cutting and Joost Noppen

School of Computing Science
University of East Anglia
Norwich, Norfolk, UK
Email: {david.cutting,j.noppen}@uea.ac.uk

***Abstract*—Various tools exist to reverse engineer software source code and generate design information, such as UML projections. Each has specific strengths and weaknesses, however no standardised benchmark exists that can be used to evaluate and compare their performance and effectiveness in a systematic manner. To facilitate such comparison we introduce the Reverse Engineering to Design Benchmark (RED-BM), which consists of a comprehensive set of Java-based targets for reverse engineering and a formal set of performance measures with which tools and approaches can be analysed and ranked. When used to evaluate 12 industry standard tools performance figures range from 8.82% to 100% demonstrating the ability of the benchmark to differentiate between tools. Most reverse engineering tools can provide their output in the Extensible Metadata Information (XMI) format. Theoretically this should ensure tool interoperability but in practice the implementation of the XMI standard varies widely to the point where outputs cannot be exchanged between tools. In addition, this severely hinders the systematic usage of reverse engineering tool output, for example in a benchmark or for use in other analysis. To aid the comparison, analysis and further use of reverse engineering XMI output we have developed a parser which can interpret the XMI output format of the most commonly used reverse engineering applications, and is used in a number of tools. These tools offer the facility for standalone examination of one or more XMI files, comparison between outputs for benchmarking or measurement, the use of XMI within Eclipse to generate UML projections in UMLet, and use of reverse engineering output in combination with other sources of relationship information. Given the imperfect performance of the majority of the reverse engineering tools tested by the benchmark a future direction of research is the combination of different sources of information, multiple tool output or other data, to build a more complete and accurate picture of structural relationships within source code.**

Keywords–Reverse Engineering; Benchmarking; Tool Comparison; XMI; Software Comprehension; UML; UML Reconstruction.

I. INTRODUCTION

Reverse engineering is concerned with aiding the comprehensibility and understanding of existing software systems. With ever growing numbers of valuable but poorly documented legacy codebases within organisations reverse engineering has become increasingly important. In response, there are a wide number of reverse engineering techniques, which offer a variety in their focus from Unified Modelling Language (UML) projection to specific pattern recognition [1][2][3]. However, it is difficult to compare their effectiveness against each other, as no standard set of targets exist to support this goal over multiple approaches, a problem also found in the verification and validation of new tools and techniques [4]. Any performance evaluations which do exist are specific to an approach or technique. It is impossible, therefore to gain a comparative understanding of performance for a range tasks, or to validate new techniques or approaches. To address this gap, a benchmark of such targets, the Reverse Engineering to Design Benchmark (RED-BM) was created that can be used to compare and validate existing and new tools for reverse engineering.

The use of benchmarks as a means to provide a standardised base for empirical comparison is not new and the technique is used widely in general science and in computer science specifically. Recent examples where benchmarks have been successfully used to provide meaningful and repeatable standards include comparison of function call overheads between programming languages [5], mathematical 3D performance between Java and C++ [6], and embedded file systems [7]. Our benchmark provides the ability for such meaningful and repeatable standard comparisons in the area of reverse engineering.

Previous work reviewing reverse engineering tools has primarily focused on research tools many with the specific goal of identification of design patterns [2][3][8][9][10], clone detection [11] or a particular scientific aspect of reverse engineering, such as pattern-based recognition of software constructs [12]. A previous benchmarking approach for software reverse engineering focused on pattern detection with arbitary subjective judgements of performance provided by users [13]. The need for benchmarks within the domain of reverse engineering to help mature the discipline is also accepted [4].

To make further use of reverse engineering output, for example, between tools or for re-projection of UML, an Object Management Group (OMG) standard, the XML Metadata Interchange (XMI) format [14], is provided. XMI is a highly customisable and extensible format with many different interpretations. In practice tools therefore have a wide variation in their XMI output and exchange between reverse engineering tools, useful for interactive projection between tools without repetition of the reverse engineering process, is usually impossible. This variance in XMI format also hinders use of XMI data for further analysis outside of a reverse engineering tool, as individual tools are required for each XMI variation.

During the creation of the reverse engineering benchmark, two tools were developed which could analyse Java source code identifying contained classes, and then, check for the

 ISBN: 978-1-61208-367-4

presence of these classes within XMI output. Further work based upon the identification and analysis of variances within different reverse engineering tools' output, along with a desire to be able to integrate such output within more detailed analysis, led to the creation of a generic XMI parser (Section III). The parser solves the problem of XMI accessibility through generic use and abstract representation of structural data contained in XMI files of multiple formats. This parser is used by further tools for structural analysis or comparison as well as automated UML re-projection within Eclipse.

The remainder of this paper in organised as follows: in Section II, we introduce our benchmark, and show it's application to industry tools (Section II-E). Section III concerns our work to make further use of reverse engineering output through the development of a generic XMI Parser. Finally, Section IV summarises work to date and details our current and future direction of research.

II. The Reverse Engineering to Design Benchmark (RED-BM)

RED-BM facilitates the analysis of reverse engineering approaches based on their ability to reconstruct class diagrams of legacy software systems. This is accomplished by offering the source code of projects of differing size and complexity as well as a number of reference UML models. The benchmark provides a set of measures that facilitate the comparison of reverse engineering results, for example class detection, to reference models including a "gold standard" and a number of meta-tools to aid in the analysis of tool outputs.

The benchmark allows ranking of reverse engineering approaches by means of an overall performance measure that combines the performance of an approach with respect to a number of criteria, such as successful class or relationship detection. This overall measure is designed to be extensible through the addition of further individual measures to facilitate specific domains and problems. In addition the benchmark provides analysis results and a ranking for a set of popular reverse engineering tools which can be used as a yardstick for new approaches. Full details, models, targets, results as well as a full description of the measurement processes used can be found at [15]. Although based on Java source code, the concepts and measurements are applicable to any object-oriented language and the benchmark could be extended to include other languages.

A. Target Artefacts

Our benchmark consists of a number of target software artefacts that originate from software packages of varying size and complexity. The range of artefacts is shown in Table I where large projects are broken down into constituent components. In addition the Table contains statistics on the number of classes, sub-classes, interfaces and lines of code for each of the artefacts.

The benchmark artefact targets represent a range of complexity and architectural styles from standard Java source with simple through to high complexity targets using different paradigms, such as design patterns and presentation techniques. This enables a graduated validation of tools, as well as a progressive complexity for any new tools to test and assess their capabilities. Also, included within RED-BM are a set of *gold standards* for class and relationship detection against which tool output is measured. These standards were created by manual analysis supported by tools, as described in Section II-D.

TABLE I. Software Artefact Targets of the RED-BM

Software Target Artefact	**Main Classes**	**Sub Classes**	**Inter-faces**	**Lines of Code**
ASCII Art Example A				
Example A	7	0	0	119
ASCII Art Example B				
Example B	10	0	0	124
Eclipse				
org.eclipse.core. commands	48	1	29	3403
org.eclipse.ui.ide	33	2	6	3949
Jakarta Cactus				
org.apache.cactus	85	6	18	4563
JHotDraw				
org.jhotdraw.app	60	6	6	5119
org.jhotdraw.color	30	7	4	3267
org.jhotdraw.draw	174	51	27	19830
org.jhotdraw.geom	12	8	0	2802
org.jhotdraw.gui	81	29	8	8758
org.jhotdraw.io	3	2	0	1250
org.jhotdraw.xml	10	0	4	1155
Libre Office				
complex.writer	11	33	0	4251
org.openoffice.java. accessibility.logging	3	0	0	287
org.openoffice.java. accessibility	44	63	1	5749
All bundled code (sw + accessibility)	241	173	33	39896

Artefacts were chosen for inclusion on the basis that they provided a range of complexity in terms of lines of code and class counts, used a number of different frameworks, offered some pre-existing design information and were freely available for distribution (under an open-source licence). Two artefacts (ASCII Art Examples A and B) were created specifically for inclusion as a baseline offering a very simple starting point with full UML design and use of design patterns.

Cactus, although depreciated by the Apache Foundation, has a number of existing UML diagrams and makes use of a wide number of Java frameworks. Eclipse was included primarily owing to a very large codebase which contains a varied use of techniques. The large codebase of Eclipse also provides for the creation of additional targets without incorporating new projects. JHotDraw has good UML documentation available both from the project itself and some third-party academic projects which sought to deconstruct it manually to UML. As with Eclipse, Libre Office provides a large set of code covering different frameworks and providing for more targets if required.

B. Measuring Performance

RED-BM enables the systematic comparison and ranking of reverse engineering approaches by defining a set of *performance measures*. These measures differentiate the performance of reverse engineering approaches and are based on accepted quality measures, such as successful detection of classes and packages [16][17]. Although seemingly both trivial and essential within a reverse engineering tool, these measures provide a basic foundation for measurement to be built on, and represent the most common requirement in reverse engineering

 ISBN: 978-1-61208-367-4

for detection of structural elements. Further, as seen in Section II-E, these measures are alone capable of differentiating wide ranges of tool performance. The performance of tools with respect to a particular measure is expressed as the fraction of data that has been successfully captured. Individual measures are then used in conjunction to form a weighted compound measure of overall performance. In our benchmark we define three base measures to assess the performance of reverse engineering tools and approaches:

- **Cl:** The fraction of classes successfully detected
- **Sub:** The fraction of sub-packages successfully detected
- **Rel:** The fraction of relationships successfully detected

Each of these measures are functions that take a system to be reverse engineered *s* and a result *r* that is produced by a reverse engineering approach when applied to *s*. The formal definition of our three base measures are as follows:

$$Cl(s,r) = \frac{C(r)}{C(s)} \quad , \quad Sub(s,r) = \frac{S(r)}{S(s)} \quad , \quad Rel(s,r) = \frac{R(r)}{R(s)} \quad (1)$$

where

$C(x)$ is the number of correct classes in *x*

$S(x)$ is the number of correct (sub-)packages in *x*

$R(x)$ is the number of correct relations in *x*

The overall performance *P* of a reverse engineering approach for the benchmark is a combination of these performance measures. The results of the measures are combined by means of a weighted sum which allows users of the benchmark to adjust the relative importance of, e.g., class or relation identification. We define the overall performance of a reverse engineering approach that produces a reverse engineering result *r* for a system *s* as follows:

$$P(s,r) = \frac{w_{CL}CL(s,r) + w_{Sub}Sub(s,r) + w_{Rel}Rel}{w_{CL} + w_{Sub} + w_{Rel}} \quad (2)$$

In this function, w_{CL}, w_{Sub} and w_{Rel} are weightings that can be used to express the importance of the performance in detecting classes, (sub-)packages and relations respectively. The benchmark results presented in this article all assume that these are of equal importance: $w_{CL} = w_{Sub} = w_{Rel} = 1$.

C. Application of the Benchmark

To analyse the effectiveness of our benchmark, we apply a range of commercial and open source reverse engineering tools (shown in Table II) to each target artefact. Each of the tools is used to analyse target source code, generate UML class diagram projections (if the tool supports such projections) and export standardised XMI data files. Although the source code target artefacts used for testing are broken down into the package level for analysis, the reverse engineering process is run on the full project source code to facilitate package identification. The output produced by each of the tools is subsequently analysed and compared to the reference UML documentation using a benchmark toolchain we specifically created for comparison of class detection rates (see Section II-D). Finally, we perform a manual consistency between the standard tool output and XMI produced to identify and correct any inconsistencies where a tool had detected an element but not represented it within the generated XMI.

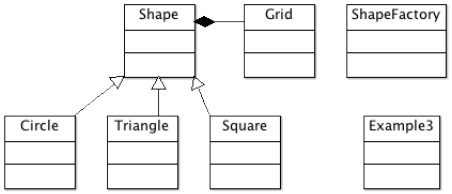

Figure 1. Reference Class Diagram Design for ASCII Art Example A

When analysing the results a wide range of variety can be observed even for simple targets such as Example A, one of the simplest targets with just 7 classes, as depicted in Figure 1. Please note that although Example A only contains generalisation and composition relationships other target artefacts contained associations, and these were included in the measurement. It can be seen in Figure 2 that Software Ideas Modeller failed to identify and display any relationship between classes. Other tools such as ArgoUML [18] (Figure 3) were very successful in reconstructing an accurate class diagram when compared to the original reference documentation.

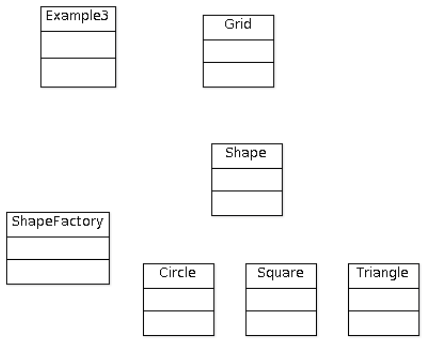

Figure 2. ASCII Art Example A Output for Software Ideas Modeller

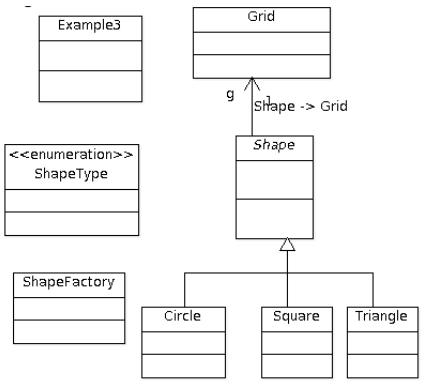

Figure 3. ASCII Art Example A Output for ArgoUML

In stark contrast to tools which performed well (e.g., Rational Rhapsody and ArgoUML) a number of tools failed

to complete reverse engineering runs of benchmark artefacts and even crashed repeatedly during this procedure. The result of which is that they are classified as detecting 0 classes for those target artefacts. While some tools failed to output valid or complete XMI data, a hindrance to their usability and ease of analysis, this has not affected their performance evaluation as their performance could be based on our manual analysis of their UML projection.

TABLE II. LIST OF TOOLS AND VERSIONS FOR USE IN EVALUATION

Tool Name (Name Used)	**Version Used (OS) Licence**
ArgoUML	0.34 (Linux) Freeware
Change Vision Astah Professional (Astah Professional)	6.6.4 (Linux) Commercial
BOUML	6.3 (Linux) Commercial
Sparx Systems Enterprise Architect (Enterprise Architect)	10.0 (Windows) Commercial
IBM Rational Rhapsody Developer for Java (Rational Rhapsody)	8.0 (Windows) Commercial
NoMagic Magicdraw UML (MagicDraw UML)	14.0.4 Beta (Windows) Commercial
Modeliosoft Modelio (Modelio)	2.2.1 (Windows) Commercial
Software Ideas Modeller	6.01.4845.43166 (Windows) Commercial
StarUML	5.0.2.1570 (Windows) Freeware
Umbrello UML Modeller (Umbrello)	2.3.4 (Linux) Freeware
Visual Paradigm for UML Professional (Visual Paradigm)	10.1 (Windows) Commercial
IBM Rational Rose Professional J Edition (Rational Rose)	7.0.0.0 (Windows) Commercial

D. Benchmark Toolchain

To facilitate effective analysis and ease reproduction or repetition of the results a toolchain was developed for use within RED-BM, consisting of two main components (*jcAnalysis* and *xmiClassFinder*), combined to measure the rate of class detection. The steps followed in the application of the benchmark are shown in Figure 4 with the developed tools highlighted.

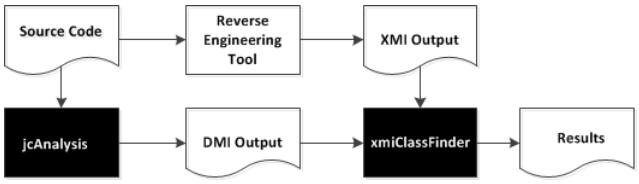

Figure 4. RED-BM Process with Toolchain Elements Highlighted

1) jcAnalysis: This tool recurses through a Java source tree analysing each file in turn to identify the package along with contained classes (primary and sub-classes). The list of classes is then output in an intermediate XML format (DMI). For every target artefact, jcAnalysis' output was compared against a number of other source code analysis utilities, including within Eclipse, to verify the class counts. A manual analysis was also performed on sections of source code to verify naming.

2) xmiClassFinder: This tool analyses an XMI file from a reverse engineering tool and attempts to simply identify all the classes contained within the XMI output (the classes detected by the reverse engineering tool in question). The classes contained within the XMI can be automatically compared to input from jcAnalysis (in DMI format) for performance (classes correctly detected) to be measured.

Once an analysis had been completed, a manual search was then performed on the source code, in XMI output, and within the reverse engineering tool itself, to try and locate classes determined as "missing" by the toolchain. This step also served to validate the toolchain, in that classes identified as "missing" were not then found to be actually present in the reverse engineering output.

E. Evaluation of Analysis Results

For the analysis of the results produced by the reverse engineering tools, we use a standard class detection performance measure for all targets (*CD*, formula 2).

To further refine the evaluation of the reverse engineering capabilities of approaches, we divide the artefacts of the benchmark into three categories of increasing complexity; C1, C2 and C3. These categories allow for a more granular analysis of tool performance at different levels of complexity. For example, a tool can be initially validated against the lowest complexity in an efficient manner only being validated against higher complexity artefacts at a later stage. Our complexity classes have the following boundaries:

- **C1:** $0 \leq$ number of classes ≤ 25
- **C2:** $26 \leq$ number of classes ≤ 200
- **C3:** $201 \leq$ number of classes

The complexity categories are based on the number of classes contained in the target artefact. As source code grows in size both in the lines of code and the number of classes it becomes inherently more complex, and so, more difficult to analyse [19][20]. While a higher number of classes does not necessarily equate to a system that is harder to reverse engineer, we have chosen this metric as it provides a quantitative measure without subjective judgement.

The bounds chosen for these categories demonstrated a noticeable drop-off in detection rates observed in many of the tools (Table III). However, any user of the benchmark can introduce additional categories and relate additional performance measures to these categories to accommodate for large scale industrial software or more specific attributes, such as design patterns.

Finally, we use the compound measure *CM*, which contains the three complexity measures with weighting as follows: $w_{C1} = 1, w_{C2} = 1.5, w_{C3} = 2$; giving a higher weighting to target artefacts that contain more lines of code.

Using these performance measures a wide range of results between the tools used for analysis can be seen. Some tools offer extremely poor performance, such as Rational Rose and Umbrello, as they crashed or reported errors during reverse engineering or UML projection, failing to detect or display classes and relationships entirely for some targets. As a general trend, the percentage of classes detected on average declined as the size of the project source code increased. As the number of classes detected varied significantly in different tools (Figure

 ISBN: 978-1-61208-367-4

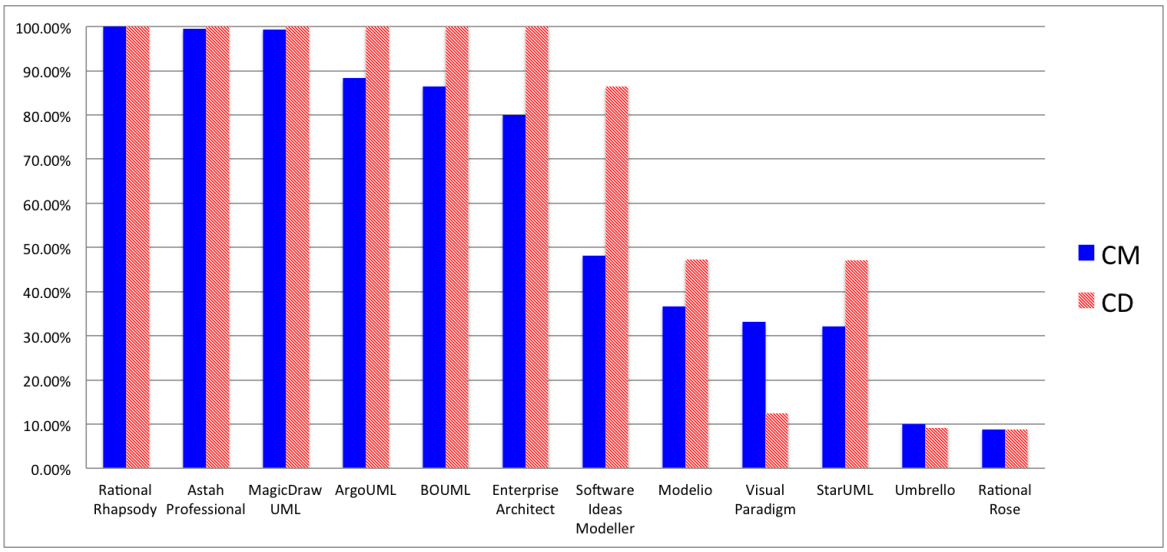

Figure 5. Overall Class Detection (CD) and Compound Measure (CM) Performance by Tool

TABLE III. CRITERIA RESULTS BY TOOL

Criterion > ∨ **Tool**	**CD** %	**C1** %	**C2** %	**C3** %	**CM** %
ArgoUML	100	98.15	75	100	88.27
Astah Professional	100	97.62	100	100	99.47
BOUML	100	92.59	75	100	86.42
Enterprise Architect	100	66.67	62.22	100	80.00
Rational Rhapsody	100	100	100	100	100.00
MagicDraw UML	100	98.15	100	100	99.38
Modelio	47.33	95.92	29.66	12.02	36.54
Software Ideas Modeller	86.41	62.15	41.48	46.04	48.10
StarUML	47.11	47.22	23.47	31.16	32.17
Umbrello	9.2	35.79	5.95	0	9.94
Visual Paradigm	12.42	38.18	51.68	16.67	33.12
Rational Rose	8.69	38.05	1.09	0	8.82

5) so did the amount of detected relationships, to a degree this can be expected as if a tool fails to find classes it would also fail to find relationships between these missing classes. In this figure the difference between the standard class detection measure CD and the compound measure CM becomes clear as, for example, ArgoUML was very strong in class detection but performed at a slightly lower level on relation detection, which is explicitly considered in the compound measure. It is also interesting to note that Visual Paradigm offered better performance for the compound measure as opposed to class detection highlighting its superior ability to deal with relations and packages as compared to class detection.

Overall our benchmark identified IBM Rational Rhapsody as the best performer as it achieved the maximum score for our compound measure (100%) with two other tools, Astah Professional and MagicDraw UML coming in a close second scoring in excess of 99%. As the poorest performers our work highlighted Umbrello, Visual Paradigm and notably IBM Rational Rose which scored the lowest with a compound measure of just 8.82% having only detected 8.69% of classes. A detailed breakdown of the performance of the tools for individual targets is provided with the benchmark [15].

III. XMI PARSER

As previously mentioned the XMI standard is highly fragmented and cannot be used as designed to interchange information between tools. It is also desirable to be able to make use of reverse engineering output for further use or analysis (for example, within a benchmark). Therefore, building from the knowledge gained in creating the toolchain for the benchmark, the simple *xmiClassFinder* tool, a XMI Parser was created.

This is a generic component designed for integration within other projects consisting of a Java package. The parser is capable of reading an XMI file, of most common output formats, recovering class and relationship information in a structured form. Data access classes are provided, which contain the loaded structural information, and can be accessed directly or recursively by third-party tools. As a self-contained utility package, the XMI Parser can be developed in isolation to tools making use of it and be incorporated into tools when required. A number of tools have been and continue to be developed within UEA to make use of reverse engineering information through implementation of the XMI Parser.

A. XMI Analyser

XMI Analyser uses the generic XMI Parser to load one or more XMI files which can then be analysed. Features include a GUI-based explorer showing the structure of the software and items linked through relationships. A batch mode can be used from the command line for automated loading of XMI files and analysis. XMI Analyser is primarily used for testing revisions to the XMI Parser, as an example application and also for the easy viewing of structural information contained within XMI, as shown in Figure 6.

XMI Analyser is also capable of comparison between multiple XMI files generating a report highlighting any differences found. This analysis can inform decisions as to the accuracy of the reverse engineering data represented in reverse engineering output.

 ISBN: 978-1-61208-367-4

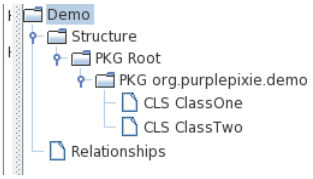

Figure 6. XMI Analyser Structure Display

B. Eclipse UMLet Integration

One of our desired outcomes was the ability to re-project UML outside of a specific reverse engineering tool. Such a capability would not only allow for detailed UML projections without access to the reverse engineering tool, but also programatic projection, for example in an interactive form. The Eclipse UMLet Integration, the interface of which is shown in Figure 7, is in the form of a plugin for the Eclipse Framework. The XMI Parser and supporting interfaces are included along with a graphical window-based interface and a visualisation component. This tool can load one or more XMI files and associate them with open or new UMLet documents. These documents can then be used to automatically generate a UML class diagram projection containing the structural elements contained within the XMI. An example of a re-projection within UMLet can be seen in Figure 8; please note, however, owing to a limitation in our UMLet API relationships are recovered but not shown.

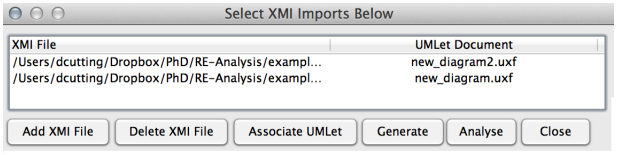

Figure 7. Eclipse Visualisation Interface

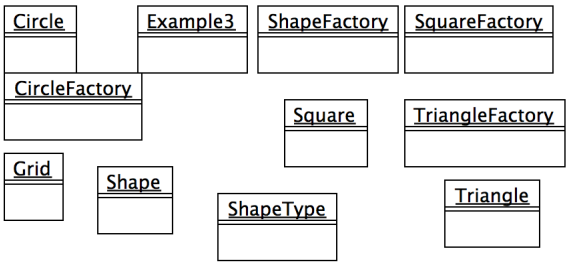

Figure 8. Eclipse UMLet Re-Projection of UML

C. Java Code Relation Analysis (jcRelationAnalysis)

The *jcRelationAnalysis* tool is a generic utility designed to analyse and comprehend the relationship between elements (classes) in Java source code. This is accomplished by first building a structural picture of the inter-relationships between elements, such as classes, contained within a source code corpus, initially from reverse engineering output, for which the XMI Parser is used. The ultimate intention of the tool is to work with combinational data from a number of different sources to compare or augment relationship information. This tool is now being used and further developed within our current and future research (Section IV).

IV. Conclusion and Future Direction

To analyse the effectiveness of RED-BM we applied it to a range of reverse engineering tools, ranging from open source to comprehensive industrial tool suites. We demonstrated that RED-BM offers complexity and depth as it identified clear differences between tool performance. In particular, using the compound measure (CM) RED-BM was capable of distinguishing and ranking tools from very low (8.82%) to perfect (100%) performance.

The XMI Parser allows tools to make direct use of reverse engineering output overcoming the fragmentation issues. The capability of direct use of reverse engineering output is clearly demonstrated through the ability for UML to be re-projected within UMLet, and also used in other tools for further analysis.

The future direction of our work will be to combine reverse engineering output with other sources of information about a source corpus, for example mining repository metadata or requirement documentation. The *jcRelationAnalysis* tool is being used as a programmable basis for integration of different sources of information into a common format of relationships between source code elements. These relationships, be they direct and found through reverse engineering, such as generalisations, or semantic in nature and found through other means, will be used in combination to form a more complete understanding of a software project.

Such analysis will aid both general comprehension of software and also change impact analysis by identifying relationships between elements not immediately obvious at the code or UML level.

References

[1] G. Rasool and D. Streitfdert, "A survey on design pattern recovery techniques," International Journal of Computing Science Issues, vol. 8, 2011, pp. 251–260.

[2] J. Roscoe, "Looking forwards to going backwards: An assessment of current reverse engineering," Current Issues in Software Engineering, 2011, pp. 1–13.

[3] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato, "A comparison of reverse engineering tools based on design pattern decomposition," in Software Engineering Conference, 2005. Proceedings. 2005 Australian. IEEE, 2005, pp. 262–269.

[4] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using benchmarking to advance research: A challenge to software engineering," in Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, 2003, pp. 74–83.

[5] A. Gaul, "Function call overhead benchmarks with matlab, octave, python, cython and c," arXiv preprint arXiv:1202.2736, 2012.

[6] L. Gherardi, D. Brugali, and D. Comotti, "A java vs. c++ performance evaluation: a 3d modeling benchmark," Simulation, Modeling, and Programming for Autonomous Robots, 2012, pp. 161–172.

[7] P. Olivier, J. Boukhobza, and E. Senn, "On benchmarking embedded linux flash file systems," arXiv preprint arXiv:1208.6391, 2012.

[8] S. Uchiyama, H. Washizaki, Y. Fukazawa, and A. Kubo, "Design pattern detection using software metrics and machine learning," in First International Workshop on Model-Driven Software Migration (MDSM 2011), 2011, p. 38.

[9] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, "An approach for reverse engineering of design patterns," Software and Systems Modeling, vol. 4, no. 1, 2005, pp. 55–70.

[10] N. Pettersson, W. Lowe, and J. Nivre, "Evaluation of accuracy in design pattern occurrence detection," Software Engineering, IEEE Transactions on, vol. 36, no. 4, 2010, pp. 575–590.

 ISBN: 978-1-61208-367-4

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," Software Engineering, IEEE Transactions on, vol. 33, no. 9, 2007, pp. 577–591.

[12] M. Meyer, "Pattern-based reengineering of software systems," in Reverse Engineering, 2006. WCRE'06. 13th Working Conference on. IEEE, 2006, pp. 305–306.

[13] L. Fulop, P. Hegedus, R. Ferenc, and T. Gyimóthy, "Towards a benchmark for evaluating reverse engineering tools," in Reverse Engineering, 2008. WCRE'08. 15th Working Conference on. IEEE, 2008, pp. 335–336.

[14] OMG et al., "Omg mof 2 xmi mapping specification," http://www.omg.org/spec/XMI/2.4.1, 2011, [Online; accessed December 2012].

[15] UEA, "Reverse engineering to design benchmark," http://www.uea.ac.uk/computing/machine-learning/traceability-forensics/reverse-engineering, 2013, [Online; accessed May 2013].

[16] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey," Journal of Software Maintenance and Evolution: Research and Practice, vol. 15, no. 2, 2003, pp. 87–109.

[17] G.-C. Roman and K. C. Cox, "A taxonomy of program visualization systems," Computer, vol. 26, no. 12, 1993, pp. 11–24.

[18] ArgoUML, "Argouml," http://argouml.tigris.org/, 2012, [Online; accessed December 2012].

[19] N. E. Fenton and S. L. Pfleeger, Software metrics: a rigorous and practical approach. PWS Publishing Co., 1998.

[20] B. Bellay and H. Gall, "A comparison of four reverse engineering tools," in Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on. IEEE, 1997, pp. 2–11.

 ISBN: 978-1-61208-367-4