An Extensible Benchmark and Tooling for Comparing Reverse Engineering Approaches

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 in this paper 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. 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.**

Keywords–Reverse Engineering; Benchmarking; Tool Comparison; Tool Support; Extensible Methods; 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. As explored in our previous work [1], 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 [2][3][4][5].

However, it is difficult to compare the effectiveness of reverse engineering techniques 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 [6]. Any performance evaluations that do exist are specific to an approach or technique. Therefore, it is impossible to gain a comparative understanding of performance for a range of tasks, or to validate new techniques or approaches. Therefore, this paper introduces a benchmark of such targets, the Reverse Engineering to Design Benchmark (RED-BM), created in order to compare and validate existing and new tools for reverse engineering.

Therefore, our goals are to:

- Create a benchmark suitable for empirical comparison of reverse engineering tools
- Apply this benchmark to current tools and evaluate the result, and their performance
- Make the benchmark extensible to support further approaches
- Provide support for extensibility through the means of data exchange between implementations

The benchmark described in this article builds on and extends previous work [1], including greater details and specifics on an inherent extensibility mechanism with an example.

The intent of the benchmark, in addition to ranking of current reverse engineering approaches, is to support further extensibility in a generic and easily accessible manner. To achieve this, a number of tools have been developed and provided which aid in the open analysis and exchange of reverse engineering output.

The remainder of this paper is organised as follows: in Section II, we introduce our benchmark before introducing the target artefacts (Section II-A) provided. Section II-B covers the performance measurements used, with Section II-C detailing how complexity is broken down for granular measurement. Extensible features of the benchmark are demonstrated in Section III, specifically the definition of new measurements (Section III-A) and use of reverse engineering output for data exchange (Section III-B). Section IV details the toolchain support for the benchmark. In Section V, the benchmark is applied against a number of industry standard tools with an evaluation of these results in Section VI and a discussion in Section VII. Related work is covered in Section VIII and the final Section IX draws a conclusion and identifies our future direction for 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 *gold standard* is a set of manually verified correct UML data, in whole or in part, for the artefacts.

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 [7]. Although based on Java source code, the core concepts and measurements, such as detection of classes, relationships, and containers, 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. These include projects such as Eclipse, an open-source integrated development environment, and LibreOffice, a large popular open-source fully-featured office package, as well as some smaller custom examples.

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 is also included as, although depreciated by the Apache Foundation, it 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.

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. The complete 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. 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 IV.

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

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 [8][9]. Although such functionality would be expected in reverse engineering tools, these measures provide a basic foundation for measurement to be built on, and represent the most common requirement in reverse engineering for detection of structural elements. Further, as seen in Section VI, 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. Therefore, these measures are built around determining the recall factor, e.g., how complete is the recovered set. 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 (successful in that a relationship was detected and is of the correct type)

Each of these measures are functions that take a system to be reverse engineered s and a reverse engineering result r (i.e., a structural UML class diagram) that is produced by a reverse engineering approach when applied to s. The formal definition of our three base measures are as follows:

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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

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: $w_{Cl} = w_{Sub} = w_{Rel} = 1$, unless mentioned otherwise.

C. Complexity Categories

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 [10], [11]. 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 for these categories were chosen as results demonstrated a noticeable drop-off in detection rates observed in the tools, as can be seen in Section VI. 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. The extensibility aspect of our work is explained in more detail in Section III.

III. Extensibility of the Benchmark

A. Extensibility of Measurements

RED-BM's included performance measures provide a solid foundation to evaluate and compare current standards of reverse engineering. To accommodate the continual advancements in this field we have made the performance measure aspect of our benchmark extensible. Any user of the benchmark can introduce new performance measures, such as the fraction of successfully detected design patterns in a given code base. Once a *gold standard* has been determined for a specific detection within the artefacts it can be tested against tool output (as explained in Section II-C for the initial criteria). With these new measures the performance of approaches can be defined for specific reverse engineering areas. In a generalised fashion we define a performance measure to be a function *M* that maps a system *s* and its reverse engineering result *r* to the domain [0..1], where 0 means the worst and 1 the best possible performance.

In addition to providing means for creating new performance measures, we provide the possibility to create new compound performance measures (i.e., measures that are compiled from a set of individual performance measures). Formally, we define a compound measure to be a function *C* that maps a system *s* and its reverse engineering result *r* to the domain [0..1], where 0 means the worst and 1 the best possible performance:

$$C(s,r) = \frac{\sum_{i=1}^{n} w_i M_i(s,r)}{\sum_{i=1}^{n} w_i} \quad (3)$$

In this expression w_i is the weighting that determines the importance of the individual performance measure *i*. Note that the performance measures we introduced in Section II-B conform to this definition and, therefore, can be seen as an example of the extensibility of the benchmark.

To further illustrate how researchers and practitioners can use this mechanism to specialise the application of RED-BM we create a performance measure that acknowledges the capability of an approach to detect design patterns during reverse engineering. This is an active research field for which to the best of our knowledge a specialised benchmark is not available.

According to literature the detection of creational and structural design patterns is easier than behavioural design patterns [12]. Therefore, we introduce two new performance measures D_b for the successful identification of creational and structural design patterns(D_{cs}), and behavioural design patterns (D_b) for a system *s* and reverse engineering result *r*:

$$D_{cs}(p,s) = \frac{P_c(r) + P_s(r)}{P_c(s) + P_s(s)} \ , \ D_b(p,s) = \frac{P_b(r)}{P_b(s)} \quad (4)$$

where

$P_c(x)$ is the number of creational design patterns in x

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE II. Simplified Comparative XMI Output from Tools

ArgoUML	**Enterprise Architect**	**Astah Professional**
<**UML:Class** xmi.id = "..." name = "Circle" visibility = "package" ...> <**UML:GeneralizableElement.generalization**> <**UML:Generalization** xmi.idref = "..." /> </**UML:GeneralizableElement.generalization**> ... <**UML:Classifier.feature** <**UML:Operation** xmi.id = "..." name = "Circle" visibility = "public" ...> </**UML:Class**>	<**packagedElement** xmi:id = "..." name = "Circle" visibility = "package" ...> <**ownedOperation** xmi:id = "..." name = "Circle" visibility = "public" ...> ... <**generalization** xmi:type = "uml:Generalization" xmi:id = "..." general = "..." > </**packagedElement**>	<**UML:Class** xmi.id = "..." name = "Circle" version = "0" ...> <**UML:ModelElement.namespace**> <**UML:Namespace** xmi.idred = "..." </**UML:Namespace** ... <**UML:ModelElement.visibility** xmi.value = "package" /> ... <**UML:GeneralizableElement.generalization**> xmi.idref = "..." /> xmi.idref = "..." /> </**UML:GeneralizableElement.generalization**>

$P_s(x)$ is the number of structural design patterns in *x*

$P_b(x)$ is the number of behavioural design patterns in *x*

In addition to these performance measures we introduce additional measures that demonstrate how to consider negative influences on performance. In this case, we consider falsely identified creational and structural design patterns (E_{cs}) and behavioural design patterns (E_b) a reverse engineering approach produces as part of the overall result:

$$E_{cs}(p, r) = 1 - \frac{F_c(r) + F_s(r)}{P_c(r) + P_s(r) + F_c(r) + F_s(r)} \quad (5)$$

$$E_b(p, r) = 1 - \frac{F_b(r)}{P_b(r) + F_b(r)} \quad (6)$$

where

$F_c(x)$ is the number of falsely identified creational design patterns in *x*

$F_s(x)$ is the number of falsely identified structural design patterns in *x*

$F_b(x)$ is the number of falsely identified behavioural design patterns in *x*

These individual performance measures for design patterns can now be combined into a single compound performance measure *DPR* for design pattern recognition in system *p* with reverse engineering result *r* that includes weightings for each individual component:

$$\mathit{DPR(p,r)} = \frac{w_{D_{cs}} D_{cs} + w_{D_b} D_b + w_{F_{cs}} F_{cs} + w_{F_b} F_b}{w_{D_{cs}} + w_{D_b} + w_{F_{cs}} + w_{F_b}} \quad (7)$$

B. Extensibility in Data Exchange

Another prominent aspect that needs to be addressed for a reusable and extensible benchmark is the gap that exists between input and output formats of various reverse engineering tools. Indeed, to make further use of reverse engineering output, for example, between tools or for re-projection of UML there is an Object Management Group (OMG) standard, the XML Metadata Interchange (XMI) format [13]. XMI is a highly customisable and extensible format with many different interpretations. Therefore, in practice, tools 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 presence of these classes within XMI output.

However, the need remained to make more generalised use of reverse engineering output XMI, beyond this specialist utility. Our research required the ability to load XMI into a memory model and manipulate and/or compare it. Additionally it was foreseen that future, as yet not specifically defined, uses could be found for programmatic access to reverse engineering output.

One of the challenges in making automated programmatic use of XMI output from different tools was the wide variety of output format. This is due to the wide range of customisation possibilities in the XMI format itself [13], it's parent Meta-Object Format (MOF; [14]), and the representation of UML elements within XMI [15]. Even the most basic structural elements such as classes, and relationships such as generalisation (inheritance) are represented in very different ways.

Such variation is shown in three XML listings showing partial example output for a class representation from ArgoUML, Enterprise Architect and Astah Professional (Table II).

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-C). 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.

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

C. XMI Parser

The XMI Parser 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.

1) 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 1.

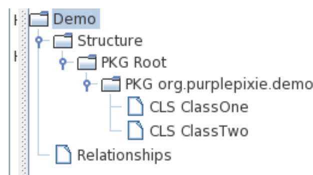

Figure 1. XMI Analyser Structure Display

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.

2) 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 2, 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 3; please note, however, owing to a limitation in our UMLet API relationships are recovered but not shown.

3) 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

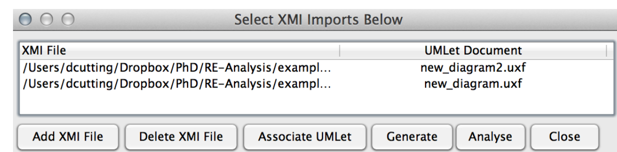

Figure 2. Eclipse Visualisation Interface

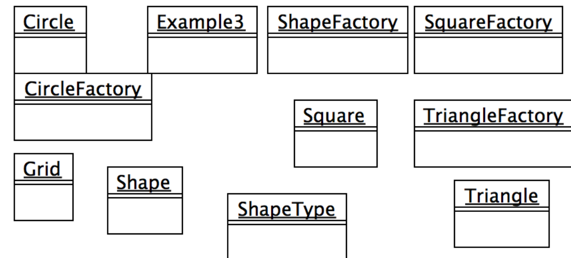

Figure 3. Eclipse UMLet Re-Projection of UML

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 IX).

IV. Benchmark Toolchain

The generic stages required to perform benchmarking are shown in Figure 4; the source code must be extracted from the project, the structural elements contained within the source code extracted directly and also by a reverse engineering tool, before the outputs are compared.

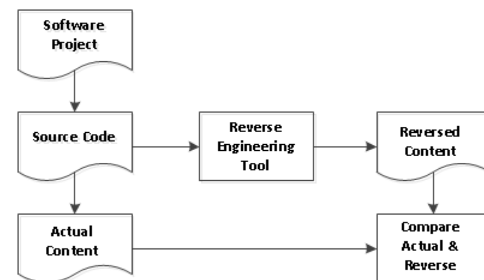

Figure 4. RED-BM Generic Process

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 5 with the developed tools highlighted.

4) jcAnalysis: This tool recurses through a Java source tree analysing each file in turn to identify the package along with contained classes (primary and nested 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

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

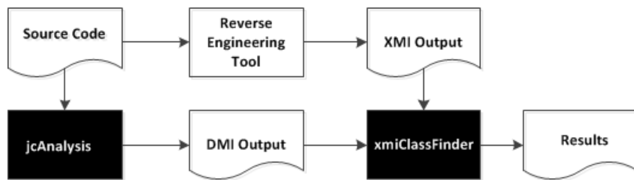

Figure 5. RED-BM Process with Toolchain Elements Highlighted

analysis was also performed on sections of source code to verify naming. Once verified, this output then constitutes the *gold standard* for class detection against which tool output is compared.

5) 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.

V. Application of the Benchmark

To analyse the effectiveness of our benchmark, we have applied a range of commercial and open source reverse engineering tools (shown in Table III) 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 generated *gold standard* using a benchmark toolchain we specifically created for comparison of class detection rates (see Section IV). 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. For this analysis we used weightings as stated, where all types of elements are of equal weight ($w_{Cl} = w_{Sub} = w_{Rel} = 1$), and categories of increased complexity have higher weight in the compound measure ($w_{C1} = 1, w_{C2} = 1.5, w_{C3} = 2$).

When analysing the results a wide range of variety was observed even for simple targets. Example A, one of the simplest targets with just 7 classes and two types of relationship, as depicted in Figure 6, demonstrates this variety. It can be seen in Figure 7 that Software Ideas Modeller failed to identify and display any relationship between classes. Other tools such as ArgoUML [16] (Figure 8) were very successful in reconstructing an accurate class diagram when compared to the original reference documentation.

TABLE III. 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

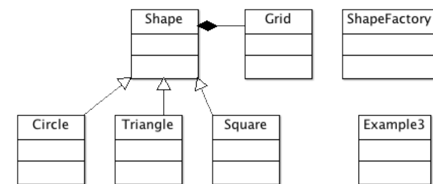

Figure 6. Reference Class Diagram Design for ASCII Art Example A

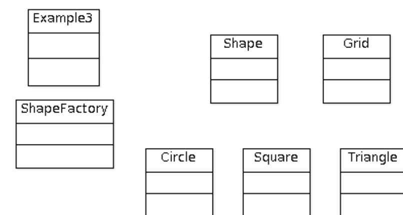

Figure 7. ASCII Art Example A Output for Software Ideas Modeller

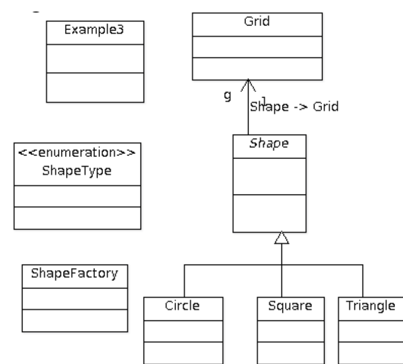

Figure 8. ASCII Art Example A Output for ArgoUML

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

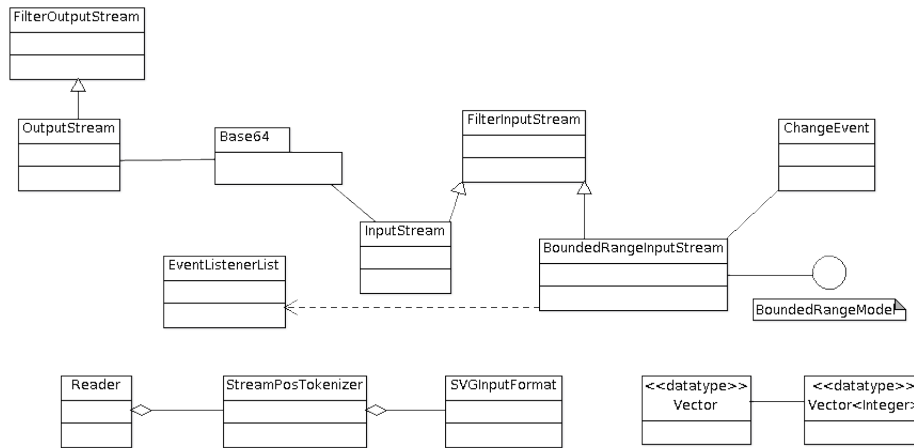

Figure 9. org.jhotdraw.io Output from Astah Professional (reconstructed)

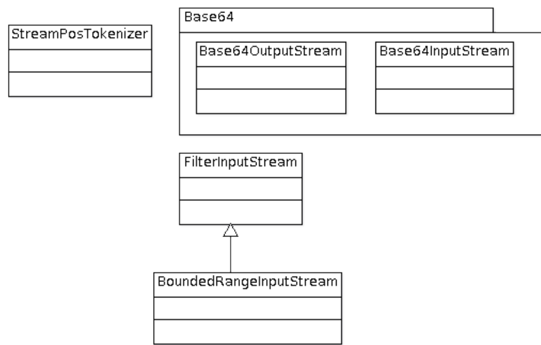

Figure 10. org.jhotdraw.io Output from Rational Rhapsody (reconstructed)

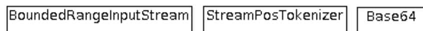

Figure 11. org.jhotdraw.io Output from ArgoUML

Another aspect in which difference is obvious relates to tool presentation, particularly when the target artefact is a Java package, which contains sub-packages nested to multiple levels. Some of the different ways tools visualise this, even for a single nesting level, is shown by the *org.jhotdraw.io* target. Tool output varies from a simple display of classes and packages at the top level (ArgoUML, Figure 11), a partial decomposition of top-level sub-packages showing contained constituent items (Rational Rhapsody, Figure 10), to a full deconstruction showing all constituent parts and relationships, but without indication of sub-package containment (Astah Professional, Figure 9).

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

TABLE IV. CRITERIA RESULTS BY TOOL

Criterion $>$ $\vee$ **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

as their performance could be based on our manual analysis of their UML projection.

VI. 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 (1)). Artefact results are broken into complexity categories as defined in Section II-C.

Finally, we use the compound measure *CM* (as defined in Section II-B, formula (3)), 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 12) 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

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

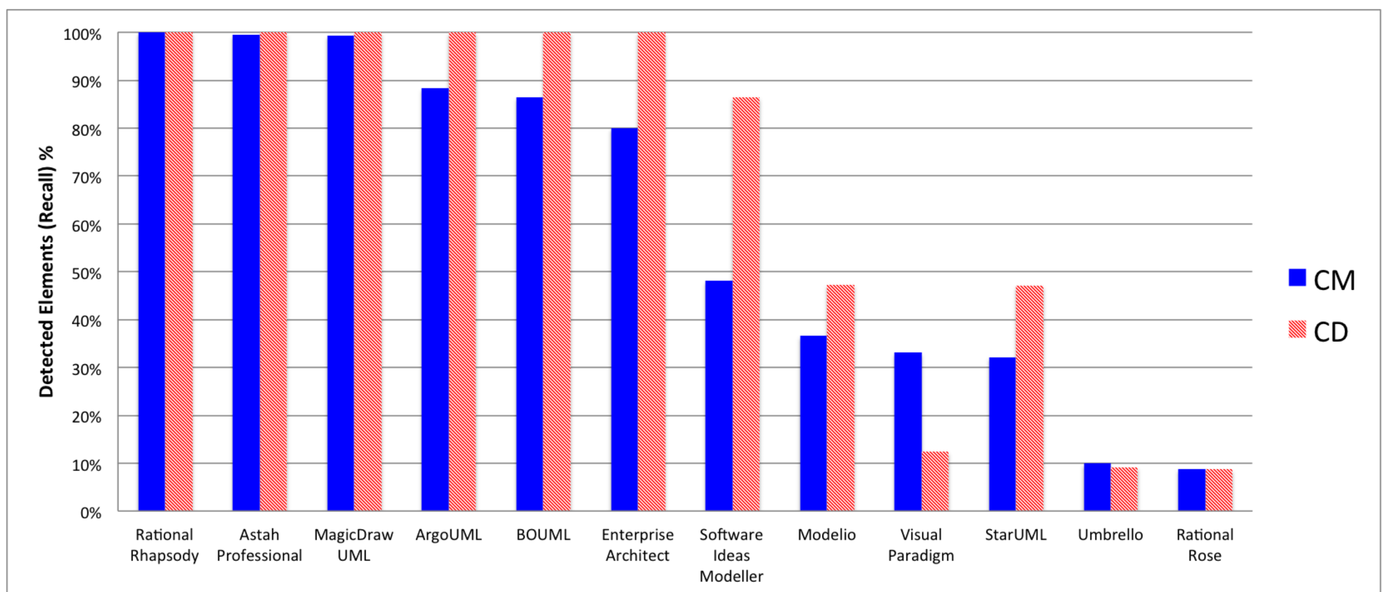

Figure 12. Overall Class Detection (CD) and Compound Measure (CM) Performance by Tool

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 [7].

This range of performance scores clearly shows a very marked differentiation between tools. At the top end some six tools score 80% or above in the compound measure, with three over 90%. In most a clear drop-off in detection rates are seen in the complexity measures as the size and complexity of the targets increase with an average measure score of 73.47%, 58.70% and 54.66% through the complexity categories C1, C2 and C3, respectively (Table IV and Figure 13).

There is a noticeable distribution of tool performance for the compound measure; five score under 40%, six score in excess of 80% and only one lies in the middle (48.1%).

It is interesting to note that of the top four performing tools three are commercial with ArgoUML, a freeware tool, scoring 88.27%. This makes ArgoUML a significantly better performer than well-known commercial solutions such Software Ideas Modeller and Rational Rose. For complete results, targets and reference documentation for this analysis please visit the benchmark website [7].

Although outside the scope of this paper, in general we found that the workflow processes of some tools were much more straightforward than others. For example, Change Vision Astah Professional and IBM Rational Rhapsody provided for straightforward generation of diagrams with configurable detail (such as optional inclusion of members and properties within class diagrams) either during or immediately after reverse engineering. On the other hand, tools such as BOUML and IBM Rational Rose required considerable manual effort in the generation of class diagrams with the need for individual classes to be placed in the diagram although relationships between classes were automatically generated. For a number of tools the lack of usability was further aggravated as their reverse engineering process repeatedly crashed or returned numerous errors on perfectly valid and compilable source code.

VII. Discussion

In the previous sections, we have demonstrated the ability of RED-BM to assess and rank reverse engineering approaches. In the following, we discuss the accuracy and validity of our approach. The targets in RED-BM currently range in size from approximately 100 to 40,000 lines of code. It can be argued that this is not representative of industrial software systems that can consist of millions of lines of code. In practice, however, reverse engineering activities tend to focus on specific areas of limited size rather than reconstructing the entire design of a very large system in a single pass [3] making our targets representative for a typical application scenario. This is supported by the capability of our benchmark to provide targets diverse enough to be able to classify the performance of a range of industry standard tools.

RED-BM currently supports the evaluation of reverse engineering approaches that focus on traditional design elements such as classes, packages and relations. It is possible that novel reverse engineering approaches will be introduced that focus on more complex design elements such as design patterns, traceability links, etc., and are beyond the current evaluation capabilities of RED-BM. However, the evaluation capabilities of the benchmark can be improved by using the extension

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

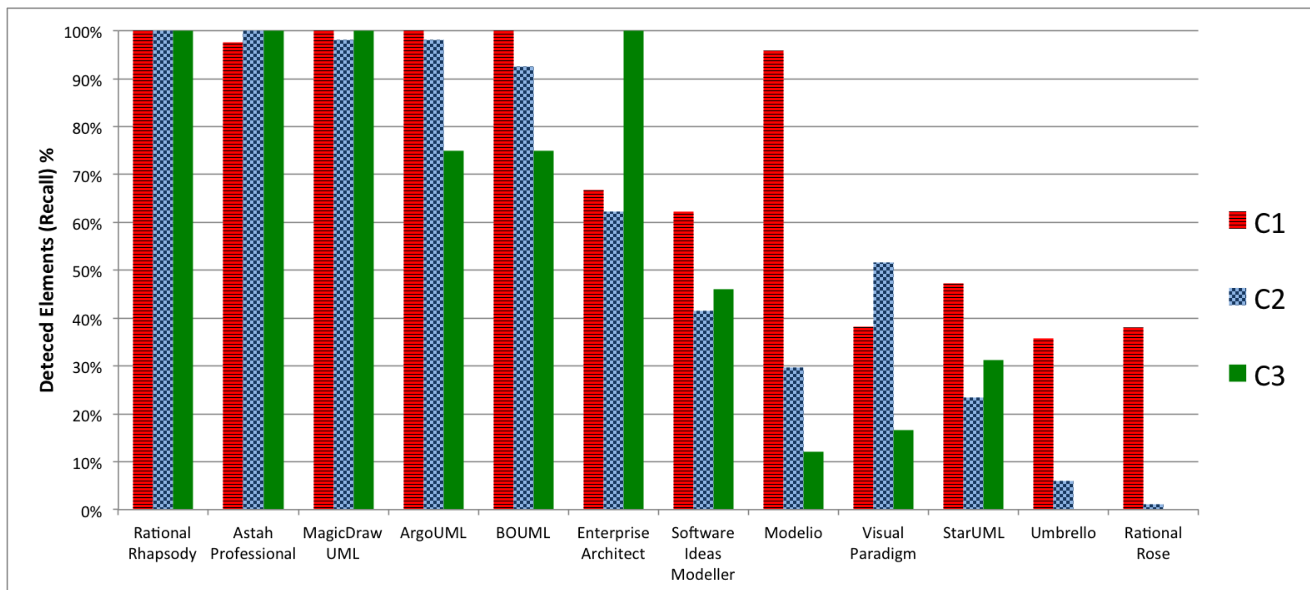

Figure 13. Tool Performance by Complexity Criteria

mechanism illustrated in Section III. Using this mechanism new performance criteria and measures can be defined that explicitly take more advanced properties of reverse engineering approaches into account.

A final point we would like to highlight is that the effort involved in evaluating tools and approaches requires the ability to efficiently compare their outputs. While there is a standardised output format available in the shape of XML Metadata Interchange (XMI) files, a wide variety of implementations exist which makes its use impractical. This can severely inhibit the application of the benchmark. To accommodate this we provide a number of utilities which can be used to a) analyse Java source code, b) analyse XMI output and c) identify components missing from the XMI output, which were found in the source code. Use of these utilities drastically reduces the time required to compare tool performance in terms of class, package and relation detection.

VIII. Related Work

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 [17], mathematical 3D performance between Java and C++ [18], and embedded file systems [19]. Our benchmark provides the ability for such meaningful and repeatable standard comparisons in the area of reverse engineering.

These previous benchmarks and others which have been reviewed (such as [20], [21], [5], and [22]) share many common features in their structure and presentation which have been incorporated into this benchmark and paper. Such is the perceived importance of benchmarks to support empirical comparison that the Standard Performance Evaluation Corporation are in the process of forming sets of standards to which benchmarks in certain areas of computer science can be created to [23].

Previous work reviewing reverse engineering tools has primarily focused on research tools many with the specific goal of identification of design patterns [3], [4], [24], [12], [25], clone detection [26] or a particular scientific aspect of reverse engineering such as pattern-based recognition of software constructs [27]. A previous benchmarking approach for software reverse engineering focused on pattern detection with arbitary subjective judgements of performance provided by users [5]. The need for benchmarks within the domain of reverse engineering to help mature the discipline is also accepted [6].

This previous work defines the importance of reverse engineering in industry as well as a research challenge. Our benchmark is a novel yet complimentary approach to previous reverse engineering benchmarks, providing a wider set of target artefacts and tool analysis than those just focused on design patterns or other specific outputs. As such it provides a solid framework for the generalised comparison of reverse engineering tools with the option of extensibility when specific measurements are required and also allows for integrating previous efforts into a single benchmark.

IX. Conclusion and Future Direction

In this paper we introduced RED BM, a benchmark for evaluating and comparing reverse engineering approaches in a uniform and reproducible manner. 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 benchmark is inherently extensible through the definition of

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

further measures, changes to weighting to shift focus, and the creation of new compound measures.

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] D. Cutting and J. Noppen, “Working with reverse engineering output for benchmarking and further use,” in Proceedings of the 9th International Conference on Software Engineering Advances. IARIA, Oct. 2014, pp. 584–590. [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=icsea_2014_21_40_10425

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

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] 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.

[12] 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.

[13] OMG et al., “OMG MOF 2 XMI Mapping Specification,” http://www.omg.org/spec/XMI/2.4.1, 2011, [Online; accessed December 2012].

[14] OMG, “OMG Meta Object Facility (MOF) Core Specification,” http://www.omg.org/spec/MOF/2.4.1, 2011, [Online; accessed December 2012].

[15] OMG, “Unified modelling language infrastructure specification,” http://www.omg.org/spec/UML/2.0/, 2005, [Online; accessed December 2012].

[16] ArgoUML, “Argouml,” http://argouml.tigris.org/, 2012, [Online; accessed December 2012].

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

[18] 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.

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

[20] Q. Quan and K.-Y. Cai, “Additive-state-decomposition-based tracking control for tora benchmark,” arXiv preprint arXiv:1211.6827, 2012.

[21] A. Bonutti, F. De Cesco, L. Di Gaspero, and A. Schaerf, “Benchmarking curriculum-based course timetabling: formulations, data formats, instances, validation, visualization, and results,” Annals of Operations Research, vol. 194, no. 1, 2012, pp. 59–70.

[22] A. Klein, A. Riazanov, M. M. Hindle, and C. J. Baker, “Benchmarking infrastructure for mutation text mining,” J. Biomedical Semantics, vol. 5, 2014, pp. 11–24.

[23] W. Bays and K.-D. Lange, “Spec: driving better benchmarks,” in Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering. ACM, 2012, pp. 249–250.

[24] 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.

[25] 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.

[26] 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.

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

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org