In Elizabeth Burd and Arie van Deursen, editors, Proc. 9th Working Conference on Reverse Engineering. IEEE, Los Alamitos, 2002

A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering

Ralf Kollmann[1], Petri Selonen[2], Eleni Stroulia[3], Tarja Systä[2], Albert Zündorf[4]
(kollmann@tzi.de, pselonen@cs.tut.fi, stroulia@cs.ualberta.ca,
tsysta@cs.tut.fi, zuendorf@upb.de)

[1]University of Bremen, Department of Computer Science, Germany
[2]Tampere University of Technology, Software Systems Laboratory, Finland
[3]University of Alberta, Computing Science Center, Canada
[4]Technical University of Braunschweig, Institute for Software, Germany

Abstract

Today, software-engineering research and industry alike recognize the need for practical tools to support reverse-engineering activities. Most of the well-known CASE-tools nowadays support reverse engineering in some way. But although the Unified Modeling Language (UML) has emerged as the de facto standard for the abstract graphical representation of object-oriented software systems, there does not yet exist a standard scheme for representing the reverse engineered models of software systems. Due to the differences in understanding and application of the UML notation and the proprietary extensions that different tools adopt, it is often difficult to ensure that model semantics remains unambiguous when working with different tools at the same time.

In this paper, we examine the capabilities of the two most successful industrial-strength CASE-tools in reverse engineering the static structure of software systems and compare them to the results produced by two academic prototypes. The comparisons are carried out both manually and automatically using a research prototype for manipulating and comparing UML models.

Keywords: UML, static reverse engineering, empirical study, tool evaluation

1 Introduction

Reverse engineering techniques have been subject to research for many years and have become quite well-known by the time. Lots of subordinate and related fields of research have developed, and especially their relations to software modeling has been recognized and elaborated, resulting in comparably new developments as round-trip engineering. In line with the recognition of a need for tool support in software modeling and development, it has become more and more common for modern CASE-tools to also support reverse engineering to a certain degree.

In most cases, the reverse engineering facilities provided by CASE-tools supporting the Unified Modeling Language (UML) [9] are limited to class diagram extraction. In the case of Java software systems, package hierarchies are usually shown as well. However, for understanding the underlying architectural decisions, a class diagram provides only limited help. Abstracting class diagrams into component diagrams and recognition of design patterns, for instance, are very desirable for real UML-based reverse engineering support.

While a class diagram shows the static information at the lowest level possible in UML, it nonetheless represents an abstraction of the actual object-oriented subject system itself. As a modeling language, UML does not tell how the model is to be implemented. More specifically, there is no one-to-one mapping between class diagram elements and the source code. For instance, composition and aggregation relationships can be implemented similarly even though they are conceptually different. Associations in general are difficult to be detected, especially for dynamically typed languages. Also, other language dependent difficulties occur: usage of abstract classes and interfaces in purely object-oriented languages (such as Java and Smalltalk) differs from their usage in hybrid languages (e.g., C++). Even the interpretation of generalization at the design level differs from inheritance at the source code level: in model inheritance, it is used for subtyping, while in the source code it is used, e.g., for subclassing.

Because of the differences in concepts at the design and implementation levels, interpretations are necessary for the extraction of class diagrams from the source code. Currently, no standard way to do that exists. Moreover, the tools do not allow the user to influence the interpretations. Instead, the interpretations are typically built in the algorithms.

These built-in interpretations are tool-dependent and can be harmful by leading to inconsistencies, misunderstandings, and limitations. However, the interpretations themselves seem to be less harmful than hiding the rationale behind them from the end user. If the CASE-tool supports round-trip engineering, the same interpretations can and should be used consistently both in reverse engineering and in code generation. The danger of misunderstandings becomes less crucial the better the user understands the interpretations and the limitations of the tool. While tools supporting code generation often allow the user to influence coding conventions, it is surprising that similar customizability is rarely supported when reverse engineering class diagrams from the source code, especially since constructing abstractions requires understanding and is thus known to be a process that should be carried out (semi-)manually.

In this paper, we aim at examination of the current situation by performing a case study and by comparing the results of the different tools. The examination also serves the CASE-tools users in understanding the built-in interpretations and limitations of the tools. We chose two popular commercial CASE-tools and two research prototypes especially targeted on UML-based reverse and round-trip engineering. The examined commercial CASE-tools are Together [17] and Rational Rose [13], and the research prototypes are IDEA [4, 6] and FUJABA [18]. The reverse engineering support of these tools was compared by using them for analyzing the subject Java software system Mathaino [3, 16]. A suite of model properties capturing important design decisions has been identified and and used as a measure for comparing the results of each tool.

In addition, the class diagrams extracted by Together, Rose, and IDEA have been stored in UML1.3/XMI1.1 format [12] and imported to a modeling tool TED [19], where automatic model comparison is carried out using a research prototype for manipulating and comparing UML models.

2 Examined Tools

Together Together supports reverse engineering of software systems developed in C++, Java, C#, VB, etc. When reverse engineering a Java program, Together constructs a tree view similar to the one produced by Rose but it also produces the UML class diagram at the same time (assuming that the user has chosen this type of diagram to start with). Together is able to reverse engineer the information from the source code (.java files), byte code (.class files), jar files, or packed zip files, but the models it extracts are not exactly the same across these types of input. The Java reverse engineer can be given instructions on the files, directories (packages), and libraries (for instance, Java foundation classes) to be examined. Furthermore, Together can provide, upon demand, a large array of metrics on the code examined (e.g. LOC, Halstead, complexity) and audits on the coding style used.

Rose Rational Rose supports reverse engineering of, e.g., C++ and Java software systems. When reverse engineering a Java program, Rose constructs a tree view that contains classes, interfaces, and association found at the highest level. Methods, variables etc. are nested under the owner classes. Rose also constructs (on demand) a class diagram representation of the extracted information and generates a default layout for it. Additionally, Rose automatically constructs a package hierarchy as a tree view. Rose is able to reverse engineer the information from the source code (.java files), byte code (.class files), jar files, or packed zip files. Quite similar to Together, the Java reverse engineering module can be given instructions on files, directories, packages, and libraries to be examined.

Idea The reverse engineering tool IDEA has been developed at University of Bremen, Germany, within the UML-AID (*Abstract Implementation and Design with the Unified Modeling Language*) project. The central subject of IDEA is the redocumentation of Java programs using the UML. Although reverse engineering of program behaviour has been addressed, too [5], the focus is on the static analysis of object-oriented structures using UML class diagrams. They are generally considered the most-employed and best-understood diagrams included in the UML.

To hold the program information in a data structure, a metamodel for the Java language has been developed, whose instances represent complete programs. A translation framework is employed to create UML models from the Java models, providing a standardized translation scheme. In several successive steps, transformations are applied to the model with the goal of creating an abstract design level representation of the examined program. These are the same features that have been applied for the study presented in this paper and include for example recognition of container classes, multiplicities and inverse associations [4].

Recent research addresses the extension of IDEA's functionality by several metric-based analyses [6]. These allow to visualize metrics graphically in the context of their underlying architecture, what makes it possible to understand the composition of metric values. The primary subject of this approach, however, is an algorithm for metric-based partitioning of large diagrams that allows selective visualization of semantically coherent diagram regions.

Fujaba Fujaba is a UML based CASE-tool that has been developed since 1998 at University of Paderborn. It supports code generation from class diagrams as well as activity diagrams, statecharts and collaboration diagrams. This allows to use UML as a kind of visual programming language for the development of full fledged applications without any manual coding. Fujaba aims to provide round-trip engineering support: if some developer or other tools (e.g. a version control system) modify the generated code and if these modifications stick to certain coding standards, then the Fujaba environment is able to analyze the changed code and to (re)create the corresponding UML specification. Again, this covers the static structure, i.e. the class diagrams, as well as the dynamic structure, i.e. the method bodies. To some extent, this round-trip engineering functionality may also be used for reverse engineering foreign code. This holds especially for class diagrams.

3 A Case Study

3.1 Examined Model Properties

Number of Classes (NOC) This is a general measure for the overall size of a software module. NOC values can be counted in various ways, depending on handling of special cases like container classes, inner classes and representation of interfaces. Therefore, high NOC values may indicate a more detailed representation.

Number of Associations (NOA) This metric measures the amount of interconnectedness in a module. However, more care has to be taken when interpreting NOA values than concerning NOC. Low NOA values may hint on an imprecise reverse engineering algorithm, but can also be the result of abstraction steps, for example recognition of inverse associations. As in the latter case, low values are considered better, NOA should be measured both before and after doing abstractions.

Types of Associations The UML supports different kinds of associations like directed, bidirectional, aggregation and composition. Additionally, the meaning of an association may be modified by applying adornments (e.g. tags or qualifiers) to its ends. In this section, we examine, which UML adornments and association kinds have been encountered, and under which conditions they have been employed in reverse engineering. This allows conclusions about the meaning that has been imposed on a feature by a particular tool.

Handling of Interfaces An interface is a specifier for the externally-visible operations of a class, component, or other classifier (including subsystems) without specification of internal structure [10]. In UML diagrams, interfaces are drawn as classifier rectangles (with a stereotype ≪Interface≫) or as circles. The interfaces are attached by a dashed generalization arrow to classifiers that support it. This indicates that the class provides (implements) all of the operations of the interface. The circle notation is used when the operations of the interface are hidden.

A class that uses or requires the operations supplied by the interface may be attached to the circle by a dashed arrow pointing to the circle. From the reverse engineering point of view, generation of such dependencies is important for understanding the usage of interfaces and for concluding component structures and dependencies (e.g., to abstract class diagrams to a component diagram). Furthermore, different ways of handling interfaces have impact on the NOC metric and possibly on the readability of the respective class diagram.

Handling of Java Collection classes Java collection classes are an implementation-specific means to handle collections (e.g. sets, list or maps) of objects. In design, such features do not appear, but are rather modeled by using association adornments like multiplicity or qualifiers. We examine, whether the reverse engineering algorithms in charge recognize Java collections and their contained types, or handle them like ordinary classes. Container resolution is important not only in design recovery, but also in metric-based analysis [6].

Multiplicities In the UML, to-many associations between objects are described by means of multiplicities. Precise information about multiplicities is difficult to derive and requires ideally dynamic analysis techniques, which are not supported currently by the tools observed here. We examine, if and to what extent multiplicities are recognized by means of the available static analysis methods.

Role Names The function of role names at association ends is comparable to that of attribute names in the sense of giving to an association between classes a meaningful descriptor, which depends on the end its attached to. Therefore in reverse engineering, role names can hold relevant additional information about the system infrastructure. We examine, whether role names are used and if, what kind of information they represent.

Inner classes Java inner classes, too, are an implementation-specific feature that hides a class definition within the specification of another class. Thus, recognizing inner classes is important to reflect the entire system structure.

	Rose	Together
Classes	84	42
Conventional	39	39
Inner classes	3 + 42	3
Interfaces	4	4
Associations	38 + 45	16
Aggregation	N/A	N/A
Multiplicities	N/A	N/A
Roles	$this\$0$ variable names	N/A

Table 1. Metrics for Rose and Together

Class Compartment Details We examine the level of details when resolving method signatures, e.g. whether parameter identifiers are resolved (in addition to the parameter type). This is especially important when analyzing the complete source code from method implementations.

4 Results from the Tools

4.1 Industrial Case tools

The metrics we collected on the behavior of Rose and Together are summarized in Table 1.

Classes Rose was able to find altogether 39 classes, 42 unnamed anonymous inner classes, and three named non-anonymous inner classes. Out of 39 conventional classes, 11 have inner classes (named or unnamed). Rose models inner classes the same way as any other classes (with rectangles); thus, the total number of classes in the class diagram was 84. The name of the (non-anonymous) inner class in the source code is used as the name of the modeled inner classes. In the case of unnamed inner classes, Rose generates numbers (in numerical order, starting from "1") to label the unnamed inner classes. For instance, one of the classes, called *MathainoMainFrame*, has 27 unnamed inner classes, named "1", "2",. . ., "27". Rose found 42 unnamed inner classes owned by 11 different classes and only three named inner classes.

Together, when applied to the "*.java" files of Mathaino, recognized the 39 classes of the core package and the three named inner classes. Interestingly enough, when applied to the "*.class" files of Mathaino, Together recognized all 45 inner classes, both anonymous and named. In both cases, the inner classes were shown on the diagram as part of the classes in which they belong (not as separate rectangles).

The name compartment of the class reverse engineered by Rose contains the name of the actual class and the name of its package. Both the attribute and operations compartments contain the names, types, and visibility (public, protected, or private) of the variables and methods, respectively. For each method, the types of parameters are also given. Together's diagrammatic representation of classes is similar.

Both Rose and Together (when applied to the "*.class" files of the project) can identify the classes of external packages on demand or if there exists a relationship (or a reference) to/from the analyzed package. We did not count the classes of the external packages.

Handling of Interfaces Rose uses a circle to illustrate interfaces in the class diagram. The (abstract) methods of the interfaces are written below the circle, separated with two horizontal lines , which is not the style recommended in UML. Together illustrates them using class rectangles with an ≪Interface≫ stereotype shown above the interface name. This notation is also available as an option in Rose.

Both Rose and Together found four interfaces in the Mathaino core package. Both connect the interfaces to the classes that support them, that is, the classes that implement the abstract methods defined in the interfaces. Both does that with a solid line (a realization relationship). Together uses a dashed line with a triangle at the end pointing to the interface (similar to the inheritance notation).

However, neither Rose nor Together were able to generate any dependencies between interfaces and the classes that use them (typically shown with a dashed arrow from a class pointing to the interface). This is an obvious limitation to understanding the roles of the interfaces. Further, interface dependencies are needed for abstracting a class diagram into a component diagram, understanding the interaction among different components, etc.

Associations The total number of associations found by Rose was 83. In Rose models, the relationship between an owner class and its inner class (named or unnamed) using an association with a fixed role name, $this\$0$. Therefore, 45 of the modeled associations are generated based on an inner class – owner class relationship. The other 38 associations model the relationship between a class and the types of variables it defines. In these associations, only the end of the association which is connected to the class representing the type of the variable is given a role name. The role is named by the variable itself. Rose does not produce multiplicities for the associations. In all cases, associations are directed.

Together did not extract any associations when applied to the "*.class" files of Mathaino. When applied to the "*.java" files, it extracted 16 associations. These associations are directional but do not specify any roles, neither do they specify multiplicities.

Neither tool generated any aggregation nor composition relationships.

Handling of Java Collection Classes Rose and Together handle container classes similarly. The container types of attributes used in the Mathaino core package (namely, *Vectors*) were not represented differently from any other attribute types; they are written (as strings) in the variable compartment of the class. That is, if a class has a variable, say *v*, the type of which is *Vector*, the variable compartment contains a string *v : java.util.Vector*.

4.2 Idea

When reverse engineering the Mathaino core package with IDEA, several transformation steps have been applied to the basic implementation level UML model to recognize the examined model properties. Descriptions of these steps are included in the following sections about the analysis results.

After doing the transformations, the pre-defined metrics suite has been calculated. Since some characteristics of the UML model change during the transformation steps, the metrics have been calculated anew after each step.

Metric Calculations For calculation of the NOC metric, both the number of classes from the mathaino core package as well as the total number of classes related to it (e.g. by UML associations) have been counted. Inner classes have been taken into account in addition to plain classes, if they were defined uniquely with a separate name. Classes from external packages were considered, if a relationship (or reference from method bodies) to the Mathaino core package existed.

In total, 42 classes have been discovered. This number is composed of 39 plain classes and three named (unique) inner classes (in total, 45 inner classes have been found, but anonymous inner classes were not considered here). Additionally, 35 external classes have been found, resulting in a total of 77 classes (not counting the anonymous inner classes).

The initial number of associations was 56. As expected, this number was reduced after each transformation step. During container resolution, two associations were discarded, because the respective contained classes were of type String, which is handled like a primitive type and not shown as an individual class (cf. section 4.2; NOA after container resolution was 54). Finally, seven association pairs have been merged to bidirectional associations, resulting in a final NOA of 47.

Handling of Interfaces For interfaces, the default UML metamodel representation as subclasses of Classifier is used. This means that they appear in the resulting class diagrams as separate interface rectangles. The number of interfaces is independent from the number of classes and no subset of the latter. Four interfaces have been found in the core package and another six external interfaces have been in use, summing up to a total of 10.

Role Names Role names are derived from the identifiers of non-primitive attributes and are shown at the target association end of directed associations. In the special case of an association depicting the relationship of an inner class to its defining class, the keyword 'this' is used at role name at the association end of the defining class. When merging directed associations, both role names are taken, resulting in an undirected association with role names at both ends.

Handling of Java Collection Classes At implementation level, containers are shown as individual classes. To reveal the true relationships between a source object and the objects stored in a container, a resolution process is employed that analyses the source code for accesses to the interface of container objects [4].

While examining the Mathaino core package, 16 container type attributes have been found, all of which were Vectors. 15 of these could be resolved. Two could not be represented graphically because the type of the contained objects was String, which is not represented in the diagram as an individual class but handled as a primitive attribute. These were rendered using the textual UML attribute notation with a multiplicity attached. Finally, one container attribute could not be resolved at all, because the source code of its containing class did not contain any invocations of the container attribute's store operation. Since IDEA examines the source code for invocations of the collection classes' interface to determine the contained type, this one was not resolvable. Because of the single remaining association, the container class Vector was kept in the diagram.

For the graphical representation of the resolved containers, the UML qualifier notation has been employed [4], which shows a qualified attribute (the vector index) at the association source end and a 0..1 multiplicity at the target end, denoting that the number of contained elements is finite. Experience with this UML notation has shown that it is not always suitable in large diagrams. The problem is that the notation consists of two parts (one at each association end), but that it is not always possible to view both ends at the same time. Having only the multiplicity end without the qualifier tends to be confusing, since only the source end, but not the target end shows gives a hint on the qualifier of the association. We have circumvented this UML-specific problem by extending the qualifier notation by a tagged value {qualified} at the target association end.

Merging Inverse Associations We discovered seven pairs of potentially inverse associations [4]. All of them were unique, that is, no ambiguities with other associations were encountered during the merging process. Therefore, we decided to merge all of them. Three different kinds of inverses have been found:

- Between independent classes. In this case, predefined pairs of role names or known relations from the design can be taken as hints for merging. When these are not available, it is only possible to analyze the role names and rule out ambiguities.
- Between class and contained class (i.e. a class whose objects are stored in a container attribute of the source class). These are considered related because of the to-many association between source and targets.
- Between class and inner class. These were considered inherently associated because of the tight relation between inner class and defining class (cf. section 4.2).

To recognize the second group of inverses, the sequence of model abstraction steps is crucial, as they can only be discovered after resolution of container classes.

Aggregation and Composition According to the UML specification documents [11] [10], aggregation is an informal feature that is not characterizable in a precise way, as would be required to recognize it from the source code. Different than composition, aggregation has a rather imprecise definition, which describes a whole-part relationship between source and target classes. The best way to recover this relation from existing programs is sufficient knowledge of the software architecture. Since we did not have a person locally available knowing the system in detail, we decided to use aggregation only in one rather clear case, namely where a role name (at the association between classes MathainoDesktopHandler and MathainoMainFrame) contained the keyword "owner" and thus suggested an aggregation relationship.

In the UML, composition defines constraints about the instances of classes, not about classes themselves. Therefore, correct recognition of composition requires dynamic analysis techniques, which are not yet supported by IDEA. However, this is subject to future work.

Multiplicities The following multiplicity values and ranges are discovered by the IDEA tool:

- '0..1' The target element is initialized somewhere in the source code (at an unknown position). It cannot be determined statically, whether an initialization will actually happen at runtime.
- '1' The target element is definitely initiated at object creation time, i.e. either in the constructor or the class initialization block.
- '*' The star multiplicity represents a container class, e.g. a Set or a Collection. When using the qualifier notation for Lists, Vectors or arrays, the multiplicity '0..1' is used together with a qualifier. The meaning of this is that every index is assigned zero or exactly one element (the list is finite).

Of these multiplicities, all but '*' have been encountered in the Mathaino core package. The star multiplicity was not used, because the only container type employed was Vector, which is represented using the aforementioned qualifier notation.

Inner Classes IDEA recognizes non-anonymous inner classes by parsing the byte code. These are represented like conventional classes, using the Java naming convention for inner class names (as UML does not include this concept). Since an association to an inner class is implicitly bidirectional (based on the way inner classes are realized in Java), this construct can be considered the pure form of an inverse association. For each inner class, an association to its defining class is shown with the role name 'this', to indicate accessibility of the defining class from within the inner class.

Class Compartment Details All standard UML class compartment details like (class name, attributes and methods) are supported at implementation detail level. For attributes, visibility, identifier, type and an optional multiplicity for arrays are shown.

Resolution of method signatures includes identifiers of parameters, which facilitates understanding of internal structural relations and metrics analyses [6]. Thus, the complete design and implementation level signature information is available in the class diagram.

4.3 Fujaba

At the time of the writing, reverse engineering the Mathaino package with FUJABA does not (yet) produce sophisticated results. Especially, FUJABA is not yet able to deal with the to-many associations in Mathaino, reasonably. According to the round-trip engineering approach, FUJABA assumes that all attributes are encapsulated with certain access methods where the method names contain the attribute name. The Mathaino system does not employ such access methods. Instead, the private container attributes are used directly within various logical methods. Due to the lack of explicit access methods, the analysis of container/Vector entry types is not even triggered. Instead, uni-directional to-one associations to the class Vector are created. Similarly, FUJABA is not yet able to merge any pair of uni-directional associations to one bi-directional association. In FUJABA, this merge relies on the existence of explicit access methods that call each other, mutually. Without explicit access methods, this feature is not even triggered.

Generally, FUJABA provides a very flexible reverse engineering mechanism, that e.g. allows the user to define application specific patterns for the detection of certain higher level design pattern elements. Unfortunately, this does not yet apply for basic reverse engineering features like association detection. Due to this case study, this will be fixed in the near future.

Classes FUJABA is able to identify the top-level classes and all explicit, named inner classes. However, FUJABA ignores anonymous inner classes since it does not use this construct in forward engineering and thus it does not expect it in reverse engineering. However, this information is provided by our parser and the FUJABA team is currently discussing, whether such anonymous inner classes should be contained in the class diagram. In addition to the Mathaino classes, FUJABA shows all non-primitive classes that are used as attribute types as the target of to-one associations. Classes used as parameter or return types or for local variables are shown optionally.

Handling of Interfaces Interface classes are shown as usual classes with the stereotype ≪Interface≫. Classes implementing the interface inherit from it. Interface usage is not depicted, although usage information is provided by the FUJABA parser. An optional incorporation of uses relationships is current work.

Role Names Role names are derived from the identifiers of non-primitive attributes and are shown at the target association end of directed associations. When merging directed associations, both role names are taken, resulting in an undirected association with role names at both ends.

Handling of Java Collection Classes FUJABA uses an adaptable list of pre-defined container classes. Attributes of this type are automatically examined for their entry type by looking for usages of their add methods. On success such container attributes are turned into to-many associations. However, due to performance reasons, currently this mechanism is triggered through a name based identification of encapsulating access methods. This heuristic did not work for the Mathaino system.

Merging Inverse Associations The merging of pairs of inverse associations is (again) based on the identification of access methods that call each other mutually. While this works great in round-trip engineering, this did not work for the Mathaino system. The FUJABA team currently considers additional heuristics.

Aggregation and Composition In FUJABA, aggregation and composition is reflected in certain "isolateYourselfAndBecomeGarbageCollected" methods, that are forwarded to contained elements in case of a composition relationship. The Mathaino system does not contain such methods.

Multiplicities Due to conceptual considerations, FUJABA does not support lower multiplicity bounds, neither for to-one nor for to-many associations, neither on forward nor on reverse engineering. Thus, non-primitive attributes are shown using a 0..1 multiplicity and container attributes with identified entry types are shown using a 0..n multiplicity.

Inner Classes Named inner classes are shown as usual classes in the class diagram. If applicable, the optionally shown package name contains the surrounding class and or method name. So far, anonymous inner classes are ignored.

Class Compartment Details All standard UML class compartment details like (class name, attributes and methods) are supported at implementation detail level. For attributes, visibility, identifier, type and an optional multiplicity for arrays are shown. Method signatures include identifiers and types of parameters. Since this easily overcrowds the class diagram, the attribute and the method compartment may be collapsed by default if they exceed a certain size. The next version of FUJABA will provide scrollers for large attribute or method departments.

4.4 Comparison of the Models on a Common Research Platform

The CASE tools examined in this paper, and in general, fall short on supporting the comparison of different UML models against each other. Some CASE tools do offer profiling utilities for measuring a predefined set of metrics, but they differ significantly from each other, making it difficult to compare the results in a uniform manner. Furthermore, none of the tools is able to deduce the semantically equivalent elements between individual UML models and compare their internal states. To compare the models produced by the tools on a common research platform, they were exported into TED [19] using an XMI bridge, where they were compared using a set of UML *model operations* of the *BMO (Basic Model Operations) toolkit* [7].

A UML model operation produces a new UML diagram on the basis of existing ones. *Set operations* comprise one fundamental category of model operations. They apply set theoretical operations (i.e. union, difference or intersection) for two diagrams of the same type, and are typically functions with signature D x D $\longrightarrow$ D, D denoting a UML diagram type. In addition to the manual analysis of the models produced by individual CASE tools and the comparison of the results by hand, set operations, together with a few customized scripts, were used for alternative automatic comparison of the UML models under a common workbench.

In this particular case study, *the BMO toolkit* was used to compare the three models produced by Rational Rose, Together, and IDEA. The models were first analyzed using a predefined metrics calculation schema, and subsequently set operations were performed on the models in order to find the possible interesting commonalities and differences between them. Even with a relatively small system as Mathaino, it becomes evident that manual comparison of the models is tedious at best, especially for larger scale studies.

Only the IDEA model (as it was saved in XMI) conformed to the manually calculated metrics. Both Rose and Together models contained model elements external to the core `kapor.ca.ualberta.cs.mathaino` package (i.e., our primary interest). Together omits the package structure, placing every model element under a common package thus making it impossible to restrict the data only to that of the Mathaino core. For example, there were 121 classes, 43 associations, and 12 interfaces in the Together model. Rose generates model elements other than those covered in this case study, namely stereotypes, comments, components (reflecting the Java package structure) etc.

The differences in metrics from different tools also point out the weaknesses of XMI and especially the different XMI export implementations of the tools. Therefore more interesting than single model metrics are the differences and properties that can be instantly spotted. The BMO tool tries to draw the attention of the user on the potentially interesting properties and evident differences.

Table 2 shows the number of counterpart classes, interfaces, operations, attributes, associations, and generalizations found between different models by BMO. The counterpart elements are decided with a set of heuristic rules based on naming conventions, relationships, and element environment. A name-based recognition technique works especially well with models produced by a revenge engineering process due to its requirements on unique identifier naming imposed by a programming language.

After a counterpart relationship has been generated, BMO allows the results of the operations to be visualized in TED. Since the different CASE-tools excel in different areas of reverse engineering, union operation can be used for combining these models into a more complete representation of the original system. Difference operation can be used for exploring the model features generated by only one of the CASE-tools, and intersection shows the minimal submodel that both the CASE-tools agree on.

	Rose Together	Rose IDEA	Together IDEA
Class	50	41	39
Interface	2	2	4
Operation	538	302	448
Attribute	437	212	341
Association	9	16	12
Generalization	6	4	12

Table 2. Common elements found by the CASE tools

The most useful operation in the context of this case study is the symmetric difference, which can be used to visually differentiate between the interpretations of different CASE tools. Symmetric difference shows the parts generated only by one tools but not by the other. The differences typically result from different capabilities and interpretations of the tools.

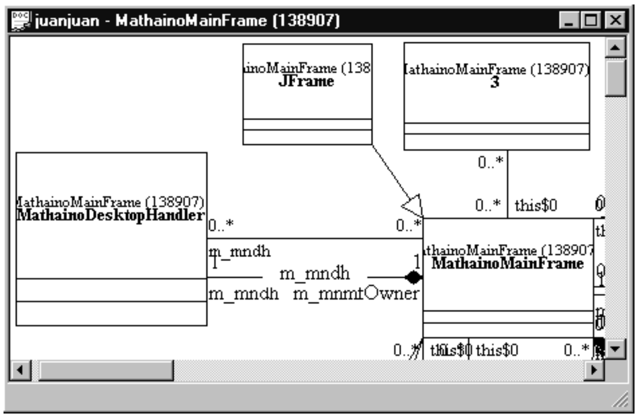

Figure 1. Comparison of the models generated by Rose and IDEA.

Figure 1 shows a portion of a TED class diagram describing the symmetric difference of IDEA and Rose models, centered on the MathainoMainFrame class. TED is not able to use visual cues such as colors for differentiating the sources of different model elements, but with the help of the textual description produced by BMO, the origin of the elements can be deduced.

It is immediately evident from the picture how Rose is able to generate the (unnamed) inner classes, but IDEA generates multiplicities, association end role names and a composition relationship. The parallel associations between MathainoMainFrame and MathainoDesktopHandler classes, the upper generated by Rose and the lower by IDEA, show the different capabilities of the tools, and also reveal the potential problems when combining the results. The multiplicities 0...*, which allow any cardinalities, are due to the XMI bridge we use; the XMI exporter of Rose generates default multiplicities. The techniques described here help to pinpoint the potentially interesting differences between the models, and in the context of this case study, the capabilities of the tools themselves.

4.5 Comparison of the Results

We compared the results of the examined tools in two different categories: basic and advanced concepts. Basic concepts refer to the UML core elements like classes and associations, and demands that the results are a valid representation of the underlying software module (i.e. no central elements have been omitted or represented imprecisely). The second category evaluates the tools' capability to generate a more abstract representation rather than a plain implementation level view. This includes strategies for design recovery and recognition of facts that are not immediately visible from the source code.

Basic concepts All of the examined tools succeeded in recognizing the basic UML features like classes, interfaces and associations. Only in one case Together failed to recognize a part of the plain associations. At this level, the results could be compared easily using metrics like NOC and NOA. The numbers were generally identical for all tools and the occurring differences could be explained by the different approaches or ways to count. For example, anonymous inner classes and package-external classes are handled differently by each tool. Similar differences existed for associations. Concluding, the creation of a correct implementation-view representation could be handled by all four tools.

Advanced concepts In the second, advanced comparison, it could be seen that the reverse engineering capabilities of the industrial tools do not go far beyond the basic UML features. More abstract representations and recognition of advanced features are clearly the domain of the research tools: These managed to recognize all of the features from the property suite either completely or at least to a certain degree, while the industrial tools did not address several of them at all (for example, multiplicities, inverse associations and container resolution have not been addressed by the industrial tools).

This leaves the impression that understanding and application of the UML in reverse engineering is still at a rather low level in industry. Our observation is underpinned by the fact that when comparing different tool versions from the previous two years, no major advancements could be found for the reverse engineering modules of the industrial tools.

The advanced concepts are semantically more challenging to be concluded than the basic concepts. This means that they provide better support for the user in understanding the software. On the other hand, it means that interpretations are needed. Therefore, the generation of the advanced concepts should be subjected to user acceptance. In a desirable case, the user involvement is supported either by allowing the user to configure the interpretations or by a providing facilities for incremental generation of different concepts (as done, e.g., in IDEA).

5 Related Work

Various empirical studies on comparisons of reverse engineering, program comprehension, and information extracting tools have been presented [2, 8, 1, 14].

Bellay and Gall presented a study in which they compared four reverse engineering tools by applying them to a commercial embedded software system, written in C [2]. They aimed at pointing out the differences in capabilities and identifying their strengths and weaknesses, especially considering their usability, extensibility, and applicability for analyzing embedded software systems. The tools used in the study vary significantly in terms of notations used. In fact, some of the tools did not have any graphical representation on the information extracted. Similar studies have been reported by Storey in [15]. In our case study, in turn, the models used are standard, namely, UML. We focused on the interpretations between the source code and the class diagram models, rather than tried to draw conclusions on the overall capabilities of the tools. The tools in our study are mostly used for software development. The reverse engineering facilities are provided mainly to support round-trip engineering. This is natural, since reverse engineering techniques are and should be used as a part of the software development process as well. Therefore, our study aims at supporting the tool users to understand the interpretations and limitations not only when an unknown software system is analyzed, but also when a familiar OO system is developed.

Armstrong and Trudeau compared traditional reverse engineering tools in their capability and applicability in architecture recovery [1]. They considered information extraction, classification, and visualization. Armstrong and Trudeau found differences in the capabilities of the tools in extracting information, namely, in parsing C code. This is in line with the study by Murphy *et al.* [8], in which three source code parsers were compared with respect to their capabilities in call graph extraction. The classification facilities in the tools are implemented to support construction of high-level models and at analyzing the constructed models in general. This is understood to be a manual or semi-automated process in reverse engineering. The classification, as well as the visualization, was supported differently in the tools analyzed. Generating a class diagram for an OO subject system can be seen to contain all these three aspects: the results of parsing are visualized as a model more abstract than the information captured in the source code. It is worth noticing that this process is carried out automatically in the tools used in this case study. Since the mapping between the code and the class diagram is not straightforward, hard-wired interpretations have been implemented in the algorithms used. On the other hand, the class diagram itself still describes information at a rather low level of abstraction, decreasing the severity of the interpretations. Also, since in round-trip engineering the code is often generated automatically, the same interpretations used for that can also be used when generating class diagrams from the code.

6 Summary and Conclusion

Reverse engineering of UML models for the subject object-oriented software system can be carried out roughly according two principles: (1) pulling out as much information as possible from the subject system and modeling it somehow using UML models or (2) aiming at design level models that are populated with the source code information whenever it is convenient. To some extent, the former resembles the traditional bottom-up reverse engineering, while the latter has a top-down flavor.

The motivation for this paper was to find out to what extent the UML is used or applicable in tool-supported reverse engineering. In the presented study, four tools from both industry and research have been compared concerning their reverse engineering capabilities. We carried out both manual and automated comparisons. The manual comparison is needed to understand the interpretations and mappings used to generate a class diagram. With automatic comparison, in turn, metrics data as well as differences and similarities between models can be quickly and easily found out. Using a standardized notation such as UML for the representation of software models makes it possible to successfully exploit model manipulation operations, such as set operations, for model analysis and comparison also with reverse engineered models. It is very difficult, if not impossible, to build such a workbench for empirical evaluations of traditional reverse engineering tools because of the notational differences. Even though in this study we focused on class diagrams, similar model operations can be built, e.g., for high-level component diagrams. The bottleneck in this approach is the exchange format: even though XMI 1.1 has its severe limitations, it is currently the only common exchange format supported by the UML-based CASE-tool vendors.

Our examination shows that although all tools provide a reliable reverse engineering functionality, only the research prototypes provided algorithms for advanced analyses. The focus in the reverse engineering facilities of the industrial tools, in turn, seem to be on the UML core features. Despite of the constantly ongoing development, the advanced reverse engineering strategies have not been considered in these tools.

Concerning the industrial tools, one crucial problem is matureness of UML support in general. They do not support the UML notation in its entirety. For example, Together has only limited support for association classes and qualifiers, and both Rose and Together do not support n-ary associations. Although Together is easily extensible via its OpenAPI and Rose via its Rose Extensibility Interface (REI), the underlying data model contains lots of simplifications that make it hard (and sometimes impossible) to reflect the full richness of the UML metamodel.

References

[1] M. Armstrong and C. Trudeau. Evaluating Architectural Extractors. In *5th Working Conference on Reverse Engineering*, pages 30–39. Hawaii, USA, 1998.

[2] B. Bellay and H. Gall. A comparison of four reverse engineering tools. In *4th Working Conference on Reverse Engineering*, pages 2–11. The Netherlands, 1997.

[3] R. V. Kapoor and E. Stroulia. Mathaino: Simultaneous Legacy Interface Migration to Multiple Platforms. In *9th International Conference on Human-Computer Interaction*, pages (vol. 1)51–55. Lawrence Erlbaum Associates, 5-10 August 2001, New Orleans, LA, USA, 2001.

[4] R. Kollmann and M. Gogolla. Application of UML Associations and Their Adornments in Design Recovery. In P. Aiken and E. Burd, editors, *Proc. 8th Working Conference on Reverse Engineering (WCRE)*, pages 81–90. IEEE, Los Alamitos, 2001.

[5] R. Kollmann and M. Gogolla. Capturing Dynamic Program Behaviour with UML Collaboration Diagrams. In P. Sousa and J. Ebert, editors, *Proc. 5th European Conference on Software Maintenance and Reengineering*, pages 58–67. IEEE, Los Alamitos, 2001.

[6] R. Kollmann and M. Gogolla. Metric-Based Selective Representation of UML Diagrams. In T. Gyimóthy and F. B. e Abreu, editors, *6th European Conference on Software Maintenance and Reengineering*. IEEE, Los Alamitos, 2002. Best Paper Award.

[7] J. Koskinen, J. Peltonen, P. Selonen, T. Systä, and K. Koskimies. Towards Tool Assisted UML Development Environments. In *7th Symposium on Programming Language and Software Tools*, 2001.

[8] G. Murphy, D. Notkin, W. Griswold, and E. Lan. An empirical study of static call graph extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, 1998.

[9] OMG, editor. *OMG Unified Modeling Language Specification, Version 1.3, June 1999*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 1999.

[10] OMG. UML Notation Guide. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [9], chapter 3.

[11] OMG. UML Semantics. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [9], chapter 2.

[12] OMG. UML Model Interchange. In OMG, editor, *OMG Unified Modeling Language Specification, Version 1.4, February 2001*, chapter 3. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 2001.

[13] Rational Software Corporation. Rose Enterprise Edition, 2002. `http://www.rational.com`.

[14] S. Sim, M.-A. Storey, and A. Winter. A Structured Demonstration of Five Program Comprehension Tools: Lessons Learnt. In *7th Working Conference on Reverse Engineering*, pages 210–212. Brisbane, Queensland, Australia, 2000.

[15] M.-A. Storey. A cognitive framework for describing and evaluating software exploration tools. Technical report, Simon Fraser University, 1998. PhD Thesis.

[16] E. Stroulia and R. V. Kapoor. Reverse Engineering Interaction Plans for Legacy Interface Migration. In *Computer Aided User-Interface Design*. Kluwer Academic, 2002 (to appear).

[17] TogetherSoft Corporation. Together 5, 2001. `http://www.togethersoft.com`.

[18] University of Paderborn. Fujaba, 2002. `http://www.fujaba.de`.

[19] J. Wikman. Evolution of a distributed repository-based architecture. Technical report, Department of Software Engineering and Computer Science, Research Report 1998:14, Blekinge Institute of Technology, Sweden, 1998. Electronic Proceedings of the First Nordic Software Architecture Workshop NOSA'98.