A Technique for Round-Trip Engineering of Behavioral UML Model Elements

Bojic, M., Dragan

Abstract—*A set of specific requirements, both functional and non-functional, to support round-trip engineering of behavioral UML model elements is derived by studying existing solutions of round-trip engineering in object-oriented modeling. This paper elaborates a use case- and test-driven approach named URCA to satisfy stated requirements. Experimental results gained with a prototype that implements the core technique are also given, along with a discussion of some questions requiring further investigation.*

Index Terms—object-oriented modeling, round-trip engineering

1. Introduction

In contemporary OO modeling tools round-trip engineering of static model elements (classes and their relationships) is supported to a reasonable level. Typical usage scenarios are: (1) building an initial model by reverse engineering program code using a static analyzer and (2) maintaining consistency between the model and the code in both directions. The first scenario is necessary if the application was initially built without modeling.

In this paper, the possible scenario of incorporating behavioral UML modeling elements in the round-trip engineering process is analyzed. The importance of having behavioral aspect of the system modeled has been discussed elsewhere, [6], [7]. Here, it is enough to state that many development methodologies, and specifically the Unified Development Process (RUP) [5] are use-case driven, and use cases are seen as an important integrating mechanism for the model as well as the process.

Proposed in a paper is a method named URCA (Use-case Driven Model Recovery by means of Formal Concept Analysis) that, in author's opinion, satisfy general requirements stated above and impose a minimal overhead to the overall round-trip engineering process. Also presented are experiences with a prototype tool that supports the core technique.

Manuscript received December 24, 2006. This work was supported in part by the German D.A.A.D. international project Software Engineering Education and Reverse Engineering.

D. M. Bojic is with the Electrical Engineering Department, University of Belgrade, Serbia (e-mail: bojic@etf.bg.ac.yu).

2. Problem Statement

Round trip engineering is defined by Wikipedia (www.wikipedia.org) as a functionality of software development tools that provides generation of models from source code (reverse engineering) and generation of source code from models; this way, existing source code can be converted into a model, subjected to software engineering methods, and them be converted back. For a mathematical treatment of this notion see e.g. [4].

Based on the definition, one can identify the following high-level usage scenarios:

1. Building the initial model from the code.
2. Updating the model from the code.
3. Updating the code from the model.

A conceptual toolset depicted in Figure 1 supports scenario-based round-trip engineering. Each element of the toolset will be explained as we elaborate high level usage scenarios.

2.1 Building the initial model from the code

The prerequisite for performing dynamic analysis of an application is to be able to compile it and run it. This is not a serious limitation, especially in the context of framework based development. The initial code is generated in the development environment, and it represents a complete application that can be compiled and executed. In any case, the user should be allowed to initially build only the static model, and to perform a complete reverse engineering once the application can be executed.

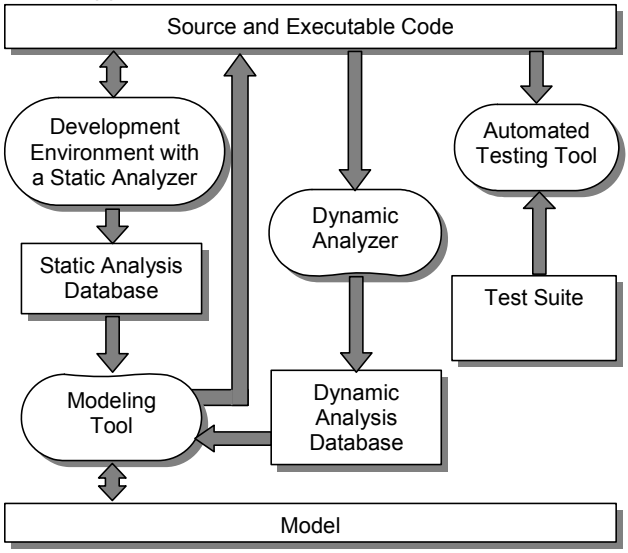

Figure 1: A toolset to support scenario-based

round-trip engineering

Now let us consider the complete process assuming that initially there is no model. The user first builds the executable code of the application. Then he or she devises a set of test cases. Some prescriptions are given in [5], but a general assumption is that for reverse engineering purposes a minimal set of test cases would be devised to relate only to the most important functionality.

Using the modeling tool, the user creates an empty Test Model as prescribed by RUP. For each test case the user updates the Test Model by adding the particular test case and one or more use cases that are traceable to that test case (only model elements for those use cases are entered, and not the relations between them). Using the testing tool, user creates a test script for each test case and enters it in the test suite. The test tool should support test scripting and should also support capture/playback, so initial script is recorded by just executing the application. The dynamic analyzer supervises every execution of the application, extracting and recording data (further discussed in section 3) in the dynamic analysis database.

When all test cases are defined and recorded, the initial model of the application is obtained "with a single click on a button", first performing static analysis to obtain static model elements, then performing processing of execution trace data to obtain behavioral elements. In the context of RUP model template, static elements are grouped in layers and packages and then placed in the Design Model. All other required RUP template elements result from processing of execution traces. These are:

- The Use Case Model, which contains actors, and abstract, concrete and included use cases. For each use case U there is a diagram "Local View" depicting all actors and all other use cases associated with U (Figure 2). There is also a diagram "Global View" representing all actors and all concrete use cases of the system.
- The Design Model, which contains use case realizations; for each realization there is a traceability use case diagram depicting <<realize>> relationship between that realization and the corresponding use case (Figure 3). Also for each realization there is a diagram of participating classes representing all classes (or their roles) that take part in a realization of the corresponding use case (Figure 4). One or more interaction diagrams are also attached to a realization, depicting possible sequences of interaction between objects that belong to that realization. The Process Model, The Business Use Case Model, the Business Objects Model, the Analysis Model and the Process Model are optional in the standard template and will not be further discussed.

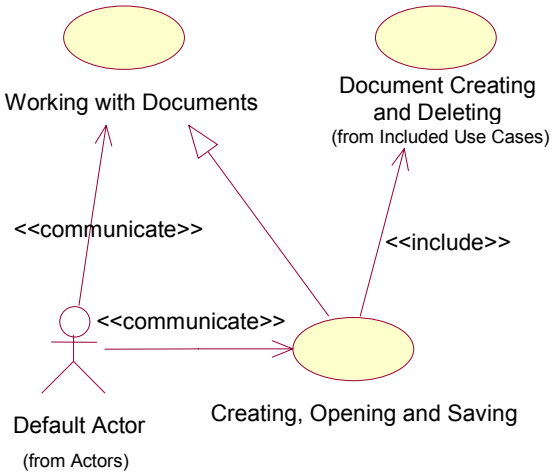

Figure 2: A sample Use Case Local View diagram

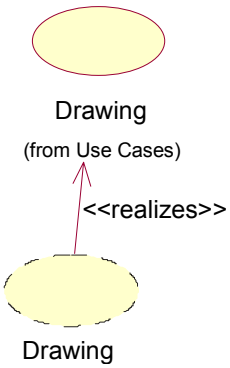

Figure 3: A sample Traceability diagram

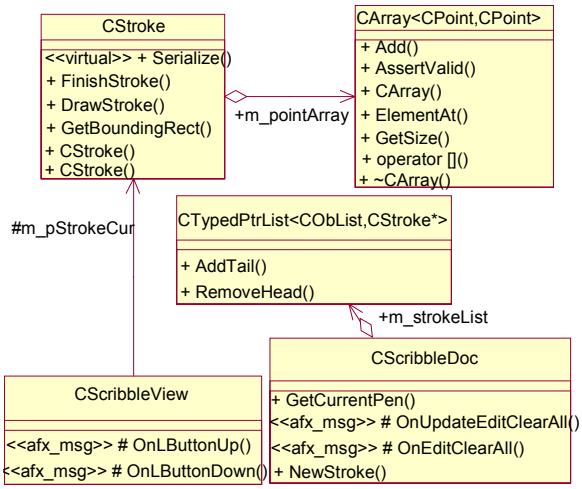

Figure 4: A sample Participating Classes diagram

2.2 Updating the model from the code

There are several concrete scenarios of updating the model from the code:

(1) The user may update only the static parts of the model, because such operation is inexpensive. The behavioral part would, of course, be left unchanged and progressively inaccurate. The inaccuracy could be partially detected by model checking (e.g. an object in an interaction diagram whose class does not exist in the model etc).

(2) Let us first introduce the term "to invalidate a test case". This term is not related to dynamic

analysis data, which are certainly outdated after the first change to the source code, but to failure of a test script to execute entirely. If a change in the source code does not invalidate any test case, and that condition can hold to a certain level of changes, then obtaining an accurate behavioral model is “one click on a button” operation, but of a long duration. The test automation tool should rerun all test scripts from the given test suite to update the dynamic analysis database. The behavioral part of the model is then built in the same way as in scenario 1, as a result processing execution trace. In an Intranet work team configuration, one can even imagine a dedicated server that constantly polls a version control server, builds a new version when it becomes available and executes test cases, thus providing a fresh dynamic analysis database to clients for a transparent update of their models.

(3) If changes in the code invalidate one or more test cases, scenario (2) can be applied to a certain extent with a graceful degradation of model preciseness. That is, test cases that fail will not be incorporated in the analysis, resulting in a model lacking certain interactions.

Ultimately, the Test Model should be updated by replacing or adding test cases and recording new scripts for those updated test cases. Dynamic analysis data should also be updated by running those tests.

2.3 Updating the code from the model

This activity is entirely driven from static elements of the model, and is not influenced in any way by behavioral elements. It should be noted that this process in general does not need any synchronizing source code comments.

Finally, a round-trip engineering feature needs to be automatic and self-configurable to the highest level possible, robust, and of acceptable performance, even transparent to the user. And that may go to the expense of approximating a model as far as it does not interfere with code generation.

3. A STUDY OF EXISTING SOLUTIONS

Let us consider, for example, how round trip engineering feature is implemented in Rational Rose, a popular UML modeling tool. The code analysis is based on static source code analysis, as well as with other main stream OO modeling tools. That means that they can accurate recover only static modeling views, like class diagrams. The main usage scenarios from our problem statement are scenarios 2 and 3, which means that the user is expected to manually create Initial application model. There is, however, support for scenario 1, but it lacks mechanisms for logically grouping model elements in subsystems, packages etc apart from physical structure of the project code. The obtained model is somewhat approximate, but it is not an obstacle because the elements that are used to regenerate code from model are obtained with sufficient correctness. During analysis, a code is populated by comments used for synchronization between the model and the code. Regenerating the model from the code and vice versa is initiated with a single command. Performance is quite satisfactory. Some other tools, notably Together, have even smoother support for round-trip engineering: there is an instant synchronization between the code and the model, without the need for a user to initiate it, and also there are no comments in code cluttering the view.

There exist several approaches to dynamically analyze the software system and produce interaction diagrams in the context of program understanding, reverse engineering, animating and debugging. However, neither of them specifically addresses round-trip engineering. What follows is a brief description of each of related methods, along with a comment on their potential strengths and weaknesses in the context of round trip engineering.

Scene [7] is a tool that enables automatic instrumentation, dynamic analysis and generation of scenario diagrams (in fact object interaction diagrams, in which messages represent object creation, function calls and returns). Scene also enables browsing documentation from a certain points in a diagram (from an object to its class declaration, from a message to a call in a source, etc, to various other documentation). The diagram abstraction is done using static and “physical” criteria: representing modules as objects, and using several operations to divide a diagram that exceeds the maximum size. One of them is call compression, that is, the exclusion from a diagram of all messages from the point of the call to the point of the corresponding return. The Scene method seems to be robust without requiring much or any preconfiguring, but lacks the notion of use case realization, and poses extra requirements on the user interface of a modeling tool.

IsVis [6] is a tool to support browsing and analyzing execution scenarios of large applications. The interesting feature is a support to the user to identify recurring patterns of interaction. Patterns are parts of interaction diagrams that can be specified by replacing some messages with a wildcard to exclude them from a matching process. The user has a visual aid for identifying patterns in the form of global overview of the complete interaction sequence named Information Mural. The tool automatically finds other occurrences of a pattern once the user specifies the first one. The collection of all such identified patterns is termed Program Model, and is used to abstract and structure interaction diagrams. Compared to URCA

approach, IsVis also uses related functionality as a grouping criterion (its patterns roughly correspond to URCA concepts), but uses manual approach to structure the model. Manual approach would make initial reverse engineering more tedious, and also impose a requirement to update the Program Model with each change in code.

SCED [12] tool illustrates an approach to automatically synthesize state diagrams from object interactions. This technique seems potentially suitable for application in round-trip engineering.

Systä [11], [13] presents an experimental Shimba tool that uses SCED and Rigi [9] to combine information obtained from static and dynamic analysis and to create various views in analyzing Java software. The event trace information included in the scenario diagrams is used to build high-level dynamic views applying two techniques: state diagrams and pattern matching. State diagrams are generated automatically for selected objects to view their overall behavior. String matching algorithms are applied to search behavioral patterns from the scenario diagrams, and present them as repetition constructs or subscenarios.

4. THE PROPOSED SOLUTION

URCA method [1], [2] is grounded on formal concept analysis [3] to group functions of the analyzed system in a hierarchical structure using a criterion of common behavior. So-called context relation is constructed on the basis of collected dynamic analysis data (function execution counts). It essentially relates each function F to a use case U if and only if there exist at least one test case that executes F at least once and is traceable to U. Dynamic analysis data are preprocessed in a suitable way:

- All irrelevant functions (essentially those that are not user defined and those that have different execution counts in two different executions of the same test case) are filtered out.
- To account for extended use cases, an operation on test data termed differentiating (essentially subtracting execution counts) is defined. Data for most test cases are for example differentiated with “app. start and exit” one.

On the basis of context relation, the concept analysis derives a concept lattice – a form of an acyclic digraph. Nodes are termed concepts. A concept is generally defined with a set of use cases and a set of related functions, and is interpreted as a UML collaboration element (that realizes those use cases). Remark: in the lattice, only one appearance of each use case (one that belongs to the highest concept) and one appearance of each function (one that belongs to the lowest concept) is taken into account, as prescribed by concept analysis. In a normal situation, there will be at most one use case per concept (in the opposite case, test cases were not discriminatory enough). Concepts that contain an empty set of use cases correspond to common functionality not “visible” from the outside, so one can define one internal function for each of those concepts.

An edge in a lattice between a pair of concepts represents generalize (in case a parent concept contains an abstract use case) or includes relations between use cases that correspond to those concepts. Extends relation is not presented in a concept lattice, but is determined on the basis on test data differing operation.

To reduce the overall number of concepts, each concept that has an empty set of user defined use cases, and the cardinality of the set of its functions below a certain threshold value, can be eliminated from the lattice. In that case, its functions are taken into account in each of successor concepts; its successors are directly related to each of their predecessors.

Figure 5 depicts the concept lattice obtained for Scribble application from Microsoft Developer Network Library. Concepts are named after the corresponding use cases, except concepts marked with #. These are for included use cases and are named after contained functions and their classes. Concept marked with □ corresponds to abstract use case, those without marks to concrete use cases. Twelve test cases in total were executed.

Actors are introduced in a model by relating a set of functions to each actor, in a similar way that IsVis [6] technique does. Manual configuration and update of this information during round trip engineering has been avoided by introducing a default actor, as explained in section 4.1.

Space limitation prevents us from explaining how each element of RUP template is generated from the concept lattice, but the following discussion will concentrate on interaction diagrams.

4.1 Generating interaction diagrams

Profiling information is, of course, not sufficient for this purpose. Therefore, URCA uses program trace in the form of function call tree, as exported from MuTek Bug Trapper tool (BT), and depicted in Figure 6. BT operation relies not on code instrumentation, but on architectural features of Intel processors. BT supports tracing debug versions of library code, providing information about parameter values and time marks, defining tracing on/off triggers in code, etc with negligible performance penalties.

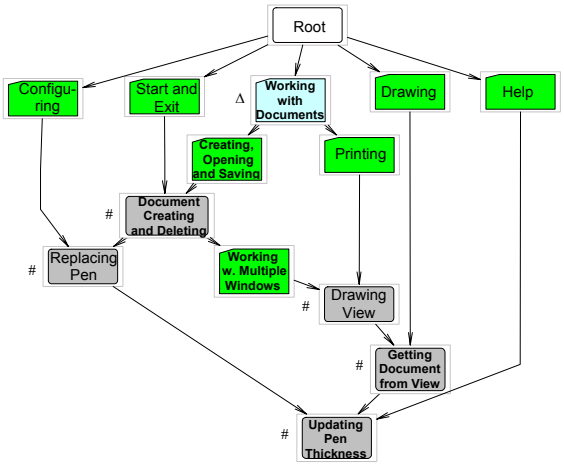

Figure 5: The concept lattice for Scribble

Nothing is required concerning the use of Bug Trapper when writing code. BT needs only the code to be compiled with debugging information included. BT provides a superset of profiler's information, therefore it is sufficient to use BT as a dynamic analyzer. Not all test cases used for constructing a lattice need to be depicted in a model with diagrams, but only those revealing the most interesting interactions.

```
void f2(){}
void f1(){ f2(); }
void f3(){ f1(); }
main() {
    int i;
    f3();
    for(i=0; i<2; i++)
        { f1(); }
    f2();
}
```

```
main( )
  f3( )
    f1( )
      f2( )
  f1( )
    f2( )
  f1( )
    f2( )
  f2( )
```

time flow

Figure 6: A sample program and its call trace

Interaction diagrams, in general, represent objects and their interactions in the form of messages – directed arcs from one object to another. Figure 7 presents an example of an interaction diagram in a form of the sequence diagram, as generated by URCA.

A collaboration diagram is another form of interaction diagram and is automatically generated from a sequence diagram and vice versa by Rose.

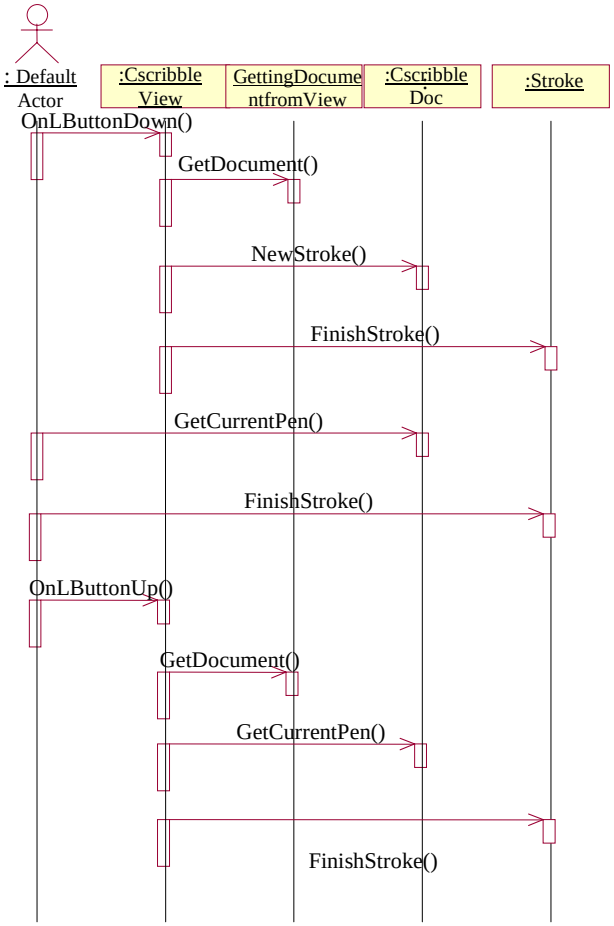

Figure 7: A sequence diagram generated by URCA

Each diagram is connected to one use case realization and the trace of one test case can generate more than one diagram. There is a discussion in [11] about abstracting interaction diagrams in a vertical way, by choosing the level of granularity at which objects are presented, and in horizontal way, by replacing a sequence of messages with a representative.

In the proposed method, diagram objects are either unique class instances, or representatives of related use case realizations (for example "Getting Document From View" object in Figure 7), or actor instances. Messages correspond to function calls, and horizontal abstraction is realized with the following operations on call traces:

• The replacement of references to every function not contained in the lattice by a communication with a default actor instance. Of course, the user can introduce more actors and assign some of those filtered out functions (and other functions) to them to obtain a more precise model. For example, Figure 8(a) and 8(b) represent traces obtained from the one in Figure 6 by filtering out f3() and main(), respectively.

• The elimination of repeating calling sequences. That is, if there exists a pair of nodes in a call tree with the same list of predecessors (starting from the root), the node that is chronologically later is eliminated if it has no

successors. The elimination of all such sequences is done in two passes through the trace, [3]. Figure 8(c) shows a call trace obtained from the one in Figure 8(b), where the calling sequence f1()-f2() has been eliminated.

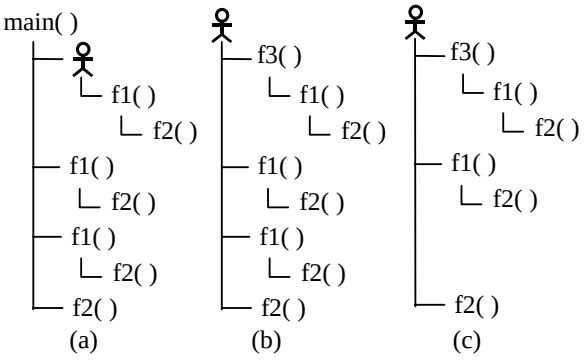

Figure 8: Operations on call traces

The principles of generating interaction diagrams based on call traces and the concept lattice are as follows: We analyze each pair of successive nodes A and B connected with a link in the call trace, and determine their corresponding concepts CA and CB, unless one or both of them are actor instances.

If CA and CB both exist and are the same, we place class instances or class utility instances for A and B on a diagram attached to that concept, and a message link between them. If CA and CB both exist but are not the same, we represent this interaction on diagrams attached to both CA and CB, placing a CB representative on a diagram attached to CA, and a CA representative on a diagram attached to CB. If one of concepts, say CA do not exist, and the other exists, then we place an actor instance A communicating with an object for B on a diagram attached to CB. If both A and B correspond to actors, we place the <<communticates>> relation between them in Global View use case diagram. Note that Rose does not provide special means to represent "included" or "extended" interactions on a diagram.

5. A Case Study

A prototype tool named URCA supports a core process (that is, reverse engineering without test automation). It runs on Windows and requires a Microsoft Visual C++ or Borland C++ profilers and MuTek Bug Trapper tool, along with Rational Rose to view the resulting model. The model is built from a script file that URCA automatically generates.

As a small but illustrative URCA usage case, let us consider the reverse engineering of UML model for Wordpad application from Microsoft development network source code examples. Wordpad consists of 925 functions (excluding MFC framework and other library functions) and about 10.000 lines of source code. 26 use cases could be devised (5 abstract and others concrete) and 22 test cases exercising them, which cover about 68% of functions. The resulting lattice contains 35 concepts, 5 for abstract use cases, 16 for concrete and 15 for included ones. Some use cases were grouped in the same concept for example "Insert line of text", "Delete line of text", and "Edit line of text" either the use cases should be reformulated or the test made more discriminatory). Distribution of functions per "included" use cases is as follows: 0 with 0 functions, 3 with 1, 4 with 2, 1 with 3, and others with more than 3 and less than 27. So we could cut a half of "included" use cases by eliminating those concepts that contain less than 3 functions. The largest potential number of objects on any interaction diagram is 12.

Another URCA usage example was a partial restructuring of XCTL — an application for controlling laboratory equipment for the analysis of semiconductor structures and their pictorial representation at the Max Planck Institute, and also a reference software reengineering project at Humboldt University Berlin. XCTL is based on a very complex problem domain and consists of about 60 custom views and dialogs, and about 44000 LOC in more than 70 C++ source files using Windows native system interface. XCTL GUI subsystem was successfully ported to Microsoft Foundation Classes framework from older technology using URCA methodology.

6. Conclusions

After establishing that a round-trip engineering feature needs to be automatic and self-configurable to the highest level possible, robust, and of acceptable performance, even transparent to the user, the URCA method is proposed in this paper as a possible solution for round-trip engineering of full models according to Unified Process template.

One of the questions that have to be further investigated is a support for consistency checking of manually created model elements, which form an abstraction of a real system (for example, the Analysis Model). The possible way to resolve a problem is to define traceability links to those elements and apply an approach similar to software reflexion modeling [10].

Another issue is defining further horizontal and vertical abstractions for behavioral model elements. The new abstractions would probably include static model elements.

References

[1] Bojić, D., Eisenbarth, T., Koschke, R., Simon, D., Velasevic, D, "Addendum to Locating Features in Source Code", *IEEE*, Transactions on Software Engineering, Vol. 30, No. 2, February 2004, p. 140

[2] Bojić , D., "An Approach to Reverse Engineering of Use Cases", PhD Thesis, *University of Belgrade, Faculty of Electrical Engineering*, Bulevar kralja Aleksandra 73, 11000 Belgrade, Yugoslavia, 2001.

[3] Burmeister , P., "Formal Concept Analysis with ConImp: Introduction to the Basic Features", *Arbeitsgruppe Allgemeine Algebra und Diskrete Mathematik,*

Technische Hochschule Darmstadt, Schloßgartenstr. 7, 64289 Darmstadt, Germany, 1998
[4] Henriksson, A., Larsson, H, "A definition of round trip engineering", Dept. of Computer and Information Science, Linköpings Universitet, SE-581 83 Linköping, Sweden, 2003.
[5] Jacobson, I., Booch, G., Rumbaugh, J., "The Unified Software Development Process", *Addison-Wesley*, 1999
[6] Jerding, D, Rugaber, S., "Using visualization for architectural localization and extraction", *IEEE*, Proc. of the 4th Working Conference on Reverse Engineering (WCRE'97), pages 56–65.
[7] Koskimies, K., Mössenböck, H. "Scene: Using scenario diagrams and active text for illustrating object-oriented programs", *IEEE*, Proc. of International Conference on Software Engineering (ICSE '96), pages 366–375.
[8] Koskinen, J., Kettunen, M, Systä T, "Profile based approach to support comprehension of Software Behavior", 1st int. conf on program comprehension, Greece, June 2006, pp. 212–224.
[9] Müller, H., Orgun, M., Tilley S.R., Uhl J.S., "A Reverse Engineering Approach To Subsystem Structure Identification", *Software Maintenance: Research and Practice*, 54., December 1993, pp. 181-204.
[10] Murphy, G., Notkin, D., Sullivan, K., "Software Reflexion Models Bridging the Gap between Source and High-Level Models", *ACM*, Proceedings of the ACM SIGSOFT '95, pp. 18-28.
[11] Systä T., "On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software", *IEEE*, Proceedings the 6th Working Conference on Reverse Engineering (WCRE99), 1999, pp. 304-313
[12] Systä, T., Koskimies, K., "Extracting state diagrams from legacy systems", ECOOP '97. pp 110-114.
[13] Systä T., "Understanding the Behavior of Java Programs", *IEEE*, Proc. of the 7th Working Conference on Reverse Engineering (WCRE 2000), ,http://www.cs.tut.fi/~tsysta/publications.html

Dragan M. Bojić (M'93) received a PhD degree in electrical engineering and computer science from the University of Belgrade in 2001. He is an assistant professor at the Faculty of Electrical Engineering, University of Belgrade. His research interests focus on software engineering techniques and tools, and e learning.