Automated Reverse-engineering of UML v2.0 Dynamic Models

Yann-Gaël Guéhéneuc
GEODES
University of Montreal
guehene@iro.umontreal.ca

Tewfik Ziadi
Triskell
IRISA-IUT Lannion
tziadi@irisa.fr

Abstract

In this position paper, we advocate the automated reverse-engineering of UML v2.0 dynamic models, i.e., sequence diagrams and statecharts, to perform high-level analyses, such as conformance checking and pattern identification. Several approaches exist to reverse-engineer UML dynamic models; However, to our best knowledge, none of these approaches consider reverse-engineering UML v2.0 dynamic models and performing high-level analyses with these models. We present our approach to UML v2.0 dynamic models reverse-engineering and sketch some use of these models. We conclude by a discussion on some issues related to the models, their reverse-engineering, and their use.

1 Introduction

Maintenance amounts for 50% of the total cost of the software development cycle. Maintainers spent more than 50% of their time understanding the programs before committing changes. To understand object-oriented programs, maintainers must use reverse-engineering techniques because documentation (text and diagrams) is often incomplete and–or obsolete.

Many works investigate the reverse-engineering of UML static models, such as class diagrams. However, there is little work on reverse-engineering dynamic models, although, in addition to static models, the UML includes notations to specify the dynamic behaviour of programs, such as sequence diagrams and statecharts.

Dynamic models of programs are as important as static models because they allow maintainers to identify complex interactions among objects and to disambiguate message sends when inheritance, delegation, polymorphism, dynamic binding, reflection are used

intensively (for example, when using design patterns such as the Abstract Factory, Observer).

This position paper proposes to go beyond static models by proposing an approach for reverse-engineering UML sequence diagrams and statecharts and for using such reverse-engineered dynamic models to perform high-level analyses. While existing work [2, 14] focus on UML v1.x sequence diagrams, our approach use UML v2.0 sequence diagrams, which include interesting composition operators. Thus, we can combine several sequence diagrams obtained from different traces of a same program to describe the *general* behaviour of the program. We obtain statecharts for objects from sequence diagrams using an existing method for statecharts synthesis [18].

Once we have generated sequence and statecharts, we describe two high-level analyses: Conformance checking and pattern identification. To our best knowledge, no existing work on dynamic model reverse-engineering perform such analyses. Also, the use of dynamic models provide interesting data to perform these analyses with respect to static models only.

The rest of the paper is organised as follows: Section 2 describes related work briefly; Section 3 introduces UML v2.0 dynamic models; Section 4 presents our approach to UML v2.0 dynamic model reverse-engineering, including building sequence diagrams and statecharts; Section 5 describes two high-level analyses that we can perform on dynamic models, *i.e.*, conformance checking and pattern identification; Finally, Section 6 concludes and introduces a discussion of dynamic model reverse-engineering.

2 Related Work

Several work exist on reverse-engineering sequence diagrams. However, it is worthy to note that many more work exist on static model reverse-engineering,

in particular class diagrams, and that dynamic model reverse-engineering is often left aside in reverse-engineering tools.

Briand *et al.* [2] propose a method to reverse engineer UML sequence diagrams from program execution traces. They execute a scenario of a program and recover a trace of the dynamic execution of the program when exercising this scenario. They use the recovered trace and a meta-model to describe UML v1.x sequence diagrams. They highlight that the amount of data recovered from executing a scenario might clutter the sequence diagram. They do not attempt to perform higher-level analyses using the reverse-engineered sequence diagrams.

Rountev *et al.* [13] describe a first algorithm to reverse-engineer UML v2.0 sequence diagrams by control-flow analysis of a program source code. They map control-flow graphs to the control-flow primitives of UML v2.0 sequence diagrams. They also introduce behaviour-preserving transformations to reduce the size (number of basic sequence diagrams, number of message sends...) of the reverse-engineering sequence diagrams. Their approach do not consider data obtained from dynamic analyses and thus is limited to the accuracy of the control-flow analysis. Also, they do not attempt to perform higher-level analyses on the reverse-engineered diagrams.

Di Lucca *et al.* [10] propose an approach to abstract business level UML diagrams from Web applications. They defined several heuristics to reverse-engineer UML class, sequence, and use-case diagrams. Their reverse-engineering techniques are based on manual analyses and on several heuristics to abstract diagrams at a high-level. They do not attempt to automate the reverse-engineering processes nor to analyse the resulting diagrams at a even higher-level.

3 UML v2.0 Dynamic Models

Dynamic models of programs are specified in the UML using two main formalisms: *sequence diagrams* (SDs) and *statecharts* (ST). While SDs capture interactions in a set of objects, statecharts represent the internal behaviours of single object. As underlined in [6], sequence diagrams are more an *inter-object* view of a program behaviour, while statecharts are an *intra-object* view of the program. We focus on sequence diagrams and statecharts as defined in UML v2.0.

UML v2.0 [12] SDs greatly enhance the previous versions of scenarios proposed in UML v1.x by incorporating some ideas available from Message Sequence Charts (MSCs) [7]. It is now possible to define and to compose a set of basic sequence diagrams using composition op-erators. Basic SDs describe a finite number of interactions in a set of objects. Figure 1 shows examples of basic SDs: The SD `SD1` describes the interactions of two objects: `a1` and `b1`. The vertical lines represent lifelines for the given objects. Interactions between objects are shown as horizontal messages, such as `m1`. Each message is defined by two events: message emission and message reception, which induce an ordering between emission and reception. Events situated on the same lifeline are ordered from top to bottom.

UML v2.0 basic SDs can be composed in composite SDs called *combined interaction* using a set of operators, *interaction operators* [12]. The four fundamental operators are: `seq`, `alt`, `loop`, and `par`. The `seq` operator specifies a weak[1] sequence between the behaviour of the two operand SDs. The `alt` operator defines a choice between a set of interaction operands, while the `loop` operator specifies an iteration of an interaction. The `par` operator defines a parallel composition. The combined SD `CombinedSD` in Figure 1 shows a combined interaction equivalent to the expression `loop(SD1 seq (SD2 par SD3))`.

4 Our Approach

Figure 2 illustrates our approach to the automated reverse-engineering of UML dynamic diagrams. Our method is defined in three mains steps: From traces to basic sequence diagrams, sequence diagrams composition, and statecharts synthesis. The following subsection discuss each step.

4.1 Building Sequence Diagrams

4.1.1 Dynamic Approach

We build SDs diagrams using dynamic analyses, in opposition to static analyses. For example, we use Caffeine [5], a tool for the dynamic analyses of Java programs, to generate traces of program executions describing the behaviours of each thread of a program. Then, we translate the trace data in several basic SDs, which we combine using the `seq`, `alt`, `loop`, and `par` operators to obtain a SD describing the dynamic behaviour of the program.

The difficulty in this approach is to combine the basic SDs appropriately as to reflect the general behaviour of the program. Also, dynamic analyses have

---

[1]UML v2.0 defines two sequence operators, "weak" and "strict", to define sequences. A strict sequence means that all events in the first SD are executed before events in the second diagram. A weak sequence means that only events on the same lifeline in the first SD are executed before events on the same lifeline in the second SD.
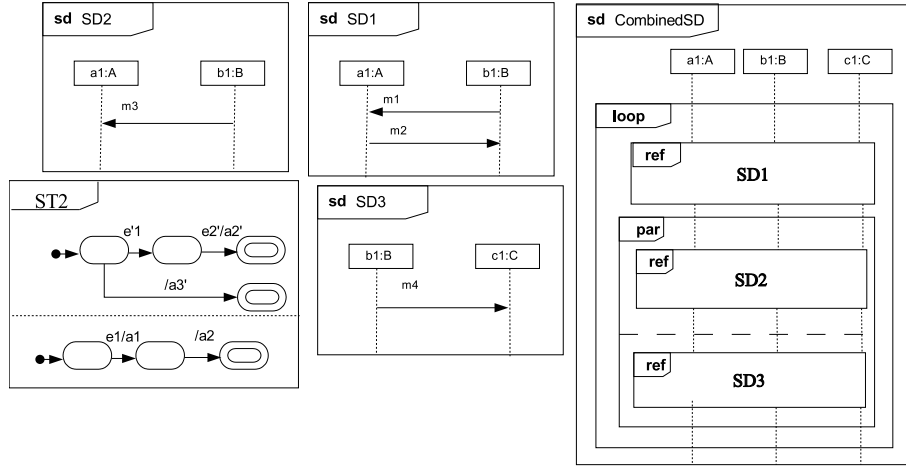
Figure 1. Examples of UML v2.0 Dynamic Models

well-known limitations, in particular accuracy with respect to the set of inputs used to generate the traces. We overcome this difficulty and the limitations by executing several time a same scenario, possibly using slightly different inputs.

4.1.2 Static Approach

Then, we complement the dynamic approach with static analyses of the programs source code. We are currently devising analyses to generate models of a program behaviour from its source code. In particular, we investigate extending and using SSA intermediate representation (such as the one produce with SOOT [15]) to built basic SDs from source code. Essentially, this approach consists in combining the work of Briand *et al.* [2] and the work of Rountev *et al.* [13], which, to our best knowledge, has never been attempted. Thus, we would get the best of static and dynamic analyses.

4.2 Building Statecharts

Once SDs have been generated, we produce statecharts automatically. SDs and STs differ by their nature: SDs capture interactions in a *set of objects*, STs represent the internal behaviour of a *single object*. Statecharts synthesis out of a collection of scenarios has received a lot of attention in the context of UML 1.x [8, 9, 11, 16]. In [18], one of the author proposed an algebraic approach to revisit the problem of statecharts synthesis in the context of UML v2.0. We propose to reuse this approach to generate UML ST from the reverse-engineered SDs in Section 4.1.

5 High-level Analyses

5.1 Conformance Checking

The first use of automatically generated SD is conformance checking. Using the models of the SD built during the design of a program and the models of SD generated automatically (by either static or dynamic analyses), we could assess their differences and highlight discrepancies.

Discrepancies can arise because the developers did not follow the design-level SD or because maintainers modified the program behaviour without updating the design-level SD. In either case, discrepancies identification would help in assessing the adequation of the current program with respect to its foreseen behaviour and either adjust its implementation or identify unforseen expected behaviours during design and update the design-level SD.

5.2 Pattern Identification

In parallel to conformance checking, we can use automatically generated SD to improve pattern identification. Patterns, either design patterns [4] or antipatterns [3], are becoming increasingly important as people recognise their impact on quality characteristics such a maintainability and understandability.

However, pattern identification is difficult because patterns involve both structural and behavioural (including creational) elements and, so far, only the structure of programs has been used to identify patterns, for examples [1, 17].
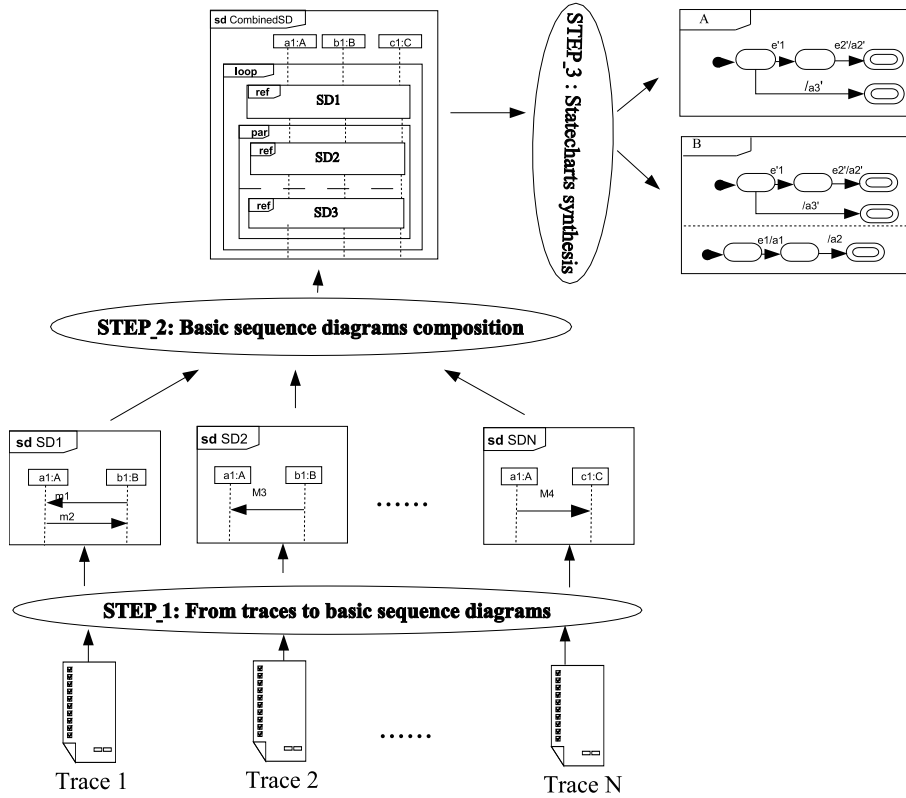
Figure 2. Our Approach: From Program Traces to UML v2.0 Dynamic Models

The use of automatically generated SD would improve pattern identification greatly by using behavioural data in addition to structural data. We can use SD either to identify "purely" behavioural patterns (and anti-patterns) or to enhance structural pattern identification by improving the speed and by reducing the number of false positives.

Structural-based search of patterns returns many false positives and is inefficient because of the many micro-architectures which structures are similar to those of patterns. The use of sequence diagrams would help in reducing the search space by removing classes which *obviously* cannot participate in a pattern because of the dynamic interactions of their instances.

Thus, the combination of class diagrams and sequence diagrams to identify patterns would lead to less false positives and decrease computation times. However, the concrete use of sequence diagrams in addition to class diagrams must be carefully studied, in particular with respect to the sequence diagrams accuracy and representation and to the exchange of data among identification algorithms based on class diagrams and those based on sequence diagrams. Also, the representation of patterns in terms of sequence diagrams constituents

for the search has not yet been studied in the literature, to our best knowledge.

6 Conclusion and Discussion

We propose to reverse-engineer sequence diagrams and statecharts automatically from a program execution traces (possibly complemented by static analyses). The objective of reverse-engineering dynamic models of a program is to perform high-level analyses, in particular conformance checking and pattern identification.

We would like to discuss with the participants the difficulty and limitations of reverse-engineering dynamic models of programs as well as the opportunities for higher-level analyses provided by such reverse-engineered dynamic models.

Acknowledgement

References

[1] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *proceedings of the 16th conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.

[2] Lionel C. Briand, Yvan Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In Eleni Stroulia and Arie van Deursen, editors, *proceedings of the 10th Working Conference on Reverse Engineering*, pages 57–66. IEEE Computer Society Press, November 2003.

[3] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1994.

[5] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In Wolfgang Emmerich and Dave Wile, editors, *proceedings of the 17th conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, September 2002.

[6] D. Harel and R. Marelly. *Come, Let's Play, Come, Let's Play Scenario-Based Programming Using LSCs and the Play-Engine*. 2003.

[7] ITU-T. Z.120 : Message sequence charts (MSC), november 1999.

[8] I. Khriss, M. Elkoutbi, and R. Keller. Automating the synthesis of uml statechart diagrams from multiple collaboration diagrams. In *Proc. of UML'98: Beyond the Notation*, pages 115–126, 1998.

[9] K. Koskimies, T. Syst, J. Tuomi, and T. Mnnist. Automated support for modeling oo software. *IEEE Software*, 15:87–94, Janu 1998.

[10] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, Porfirio Tramontana, and Ugo De Carlini. Abstracting business level UML diagrams from web applications. In Kenny Wong, editor, *proceedings of the 5th International Workshop on Web Site Evolution*, pages 12–19. IEEE Computer Society Press, September 2003.

[11] Erkki Mäkinen and Tarja Systä. MAS – An interactive synthesizer to support behavioral modelling in UML. In Mary Jean Harrold and Wilhelm Schäfer, editors, *proceedings of the 23rd International Conference on Software Engineering*, pages 15–24. IEEE Computer Society Press, May 2001.

[12] Object Management Group OMG. Uinified modeling language specification version 2.0: Superstructure. Technical Report pct/03-08-02, OMG, 2003.

[13] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Control flow analysis for reverse engineering of sequence diagrams. Technical Report OSU-CISRC-3/04-TR12, Ohio State University, March 2004.

[14] Tarja Systa. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, 2000.

[15] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot – A Java optimization framework. In *proceedings of the 9th IBM Centers for Advanced Studies Conference*, pages 125–135. ACM Press, September 1999.

[16] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceeding of International Conference on Software Engineering (ICSE 2000)*, 2000.

[17] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *proceedings of the 26th conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

[18] T. Ziadi, L. Hélouët, and J-M. Jézéquel. Revisiting statecharts synthesis with an algebraic approach. In *International Conference on Software Engineering, ICSE'26, Edinburgh, Scotland, United Kingdom*, May 2004.