Combining Static and Dynamic Analyses to Reverse-Engineer Scenario Diagrams

Yvan Labiche
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
labiche@sce.carleton.ca

Bojana Kolbah
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
kolbah@gmail.com

***Abstract*— This paper discusses reverse engineering source code to produce UML sequence diagrams, with the aim to aid program comprehension and other software life cycle activities (e.g., verification). As a first step we produce scenario diagrams using the UML sequence diagram notation. We build on previous work, now combining static and dynamic analyses of a Java software, our objective being to obtain a lightweight instrumentation and therefore disturb the software behaviour as little as possible. We extract the control flow graph from the software source code and obtain an execution trace by instrumenting and running the software. Control flow and trace information is represented as models and UML scenario diagram generation becomes a model transformation problem. Our validation shows that we indeed reduce the execution overhead inherent to dynamic analysis, without losing in terms of the quality of the reverse-engineered information, and therefore in terms of the usefulness of the approach (e.g., for program comprehension).**

Keywords-UML; Reverse engineering; Sequence diagram; Scenario diagram; Static analysis; Dynamic analysis

I. INTRODUCTION

To fully understand an existing object-oriented system, information regarding its structure and behavior is required. When no complete and consistent design model is available, one has to resort to reverse engineering to retrieve as much information as possible through static and dynamic analyses. For example, assuming one uses the Unified Modeling Language (UML) notation [4], the class, sequence, and state machine diagrams can be (partially) reverse-engineered. Besides helping comprehension, reverse engineered diagrams can help quality assurance [5, 6].

Reverse engineering the static structure (e.g., the class diagram) of an object-oriented system is already available in many UML CASE tools (e.g., Topcased, RSA, Together), although issues such as identifying the different kinds of class relationships are still considered difficult (e.g., [7]).

Reverse engineering and understanding the behavior of an object-oriented system is a different challenge. One of the main reasons is that, because of inheritance, polymorphism, and dynamic binding, it is difficult, and sometimes even impossible to know, using only the source code, the dynamic type of an object reference, and thus which methods are going to be executed. It is then difficult to follow program execution and produce a dynamic model such as a UML sequence diagram. Purely static techniques that only rely on an analysis of the source code can at best produce a control flow graph of a method, sometimes under the form of a UML sequence diagram though a better formalism could be the UML activity diagram notation. Other techniques, combining symbolic execution and source code analysis [8], face different challenges, such as identifying infeasible paths in inter-procedural control flow graphs.

It then becomes clear that executing the system and monitoring its execution is required if one wants to retrieve meaningful information and reverse-engineer dynamic models, such as UML sequence diagrams from large, complex systems [1, 9, 10]. However, the accuracy of a reverse-engineered dynamic model depends on how extensively one observes run-time behaviour. Unfortunately, the more observations, by means of instrumentation, the longer it takes to collect dynamic information, the higher the risk of disturbing behaviour and therefore the higher the risk of inaccuracies in the reverse engineered information. On the other hand, though a static analysis can present a complete picture of what could happen at run-time, it does not necessarily show what actually happens. It thus appears desirable to focus on a synergy between static and dynamic analyses [9]. We therefore build on our previous work [1, 3], which was purely dynamic, and present a new technique that combines static and dynamic information. Our objective is to reduce instrumentation as much as possible, to reduce execution times and disturb behaviour as little as possible, and compensate for the missing (dynamic) information by collecting static information. We collect execution traces and combine that information with control flow graph information to generate UML sequence diagrams. We refer here to these diagrams as *scenario diagrams* because they are incomplete sequence diagrams modeling what happens in one particular scenario instead of modeling all possible alternative scenarios for a use case. As we discuss in [1], several scenario diagrams should then be merged into a complete sequence diagram for a given use case. This requires triggering as many varied scenarios as possible through multiple executions of the system (e.g., using black-box testing techniques), and merging them into one sequence diagram. This issue is left to future work.

To formalize our approach and specify it from a logical standpoint so that it can be analyzed and compared by future

research works, we define two models (class diagrams): one for traces and another for control flow graphs; and define mapping rules between them using the Object Constraint Language (OCL) [11]. These rules are then used as specifications to implement a tool to instrument code so as to generate traces, to analyze source code to create control flow graphs, and then transform (thanks to a third party model transformation tool) an instance of the trace model and instances of control flow graphs (for several methods) into a scenario diagram, using the UML 2.1 sequence diagram notation.

The main contributions of this paper are: (1) a combination of static and dynamic data for reverse-engineering behaviour (a similar approach has been devised [10] concurrently to ours [12], though for a different purpose); (2) a precise modeling of the approach (with models and OCL mapping rules), based on model transformations; (3) our approach is one of the rare techniques that reverse engineer alternative and iterative executions; (4) case studies showing, though on systems of limited sizes (our objective here is not to address the problem of manipulating and understanding large traces [10]), that the proposed approach indeed significantly reduces instrumentation and execution overhead while providing accurate information.

This article is structured as follows. We discuss related work in section II. Our approach is detailed in sections III to V. We report on a case study in section VI. Conclusions and future research directions are provided in section VII.

II. RELATED WORK

The area of program comprehension through dynamic analysis is varied and vibrant as a 2008 systematic survey suggests [13]. The authors systematically analyzed 176 papers (out of 4,795 initially selected) published between July 1999 and June 2008 that rely on dynamic analysis to conduct program comprehension activities. We identified[1] that 19 of those papers use some kind of dynamic analysis (e.g., debugger, instrumentation of source code) to reverse engineer object collaborations, rendered under the form of a UML sequence diagram (or a similar diagram). We focus on those 19 papers as they directly relate to our work. These works collect execution information, specifically constructor, static/non static method calls (or executions), to produce UML 1.x (or 2.x) sequence diagrams (actually, scenario diagrams using the UML notation), or some kind of scenario diagram. While some of those approaches use both static and dynamic analyses, none of them actually combines both types of analysis to produce dynamic models: the static analysis is only used to generate structural diagrams (e.g., class diagram) and the dynamic analysis is only used to generate object collaborations. In some rare cases, the static analysis is used to guide the user in selecting elements of the source code to monitor during the dynamic analysis (e.g., [14]). The vast majority of those works do not reverse engineer information on alternative executions (due to control flow statements) and generated diagrams do not therefore indicate under which conditions or repetitions objects send messages. Only one past work [15], beside our previous work [1, 3], is closely related to ours, although approaches are all only dynamic. While we instrumented the source code (using aspects) to collect method executions and control flow information [1], they rely on break points (for method and control statements) being set with a debugger [15]. Other related works indicate repetitions in generated diagrams. However, they either use a simplistic heuristic to identify repetitions [16] (specifically, contiguous repeated messages are collapsed into repetitions, which does not produce an accurate diagram in general) or recognize occurrences of known interaction patterns that must be provided by the user [17, 18].

Since the 2008 systematic survey, additional related work has been published. Once again, we focus on reverse engineering object collaborations through dynamic and/or static analysis, focusing on whether the techniques rely only on a dynamic analysis, a static analysis or both. (Other characteristics of the techniques are interesting, but they are less relevant to our approach, and are therefore not discussed here.) Some approaches attempt to generate sequence diagrams using a static analysis of the source code [19], while others rely on execution traces, though through dynamic analysis only and without recovering alternatives or loops [20-22]. One recent work, which we discuss below, combines both kinds of analyses [10].

Several researchers reverse engineer sequence diagrams for web applications [23-25] or distributed systems [26]. In [23] traces are collected through purely dynamic means, and are trimmed by rejecting any new trace that is identical to an already discovered trace. The authors suggest a similar trimming approach to recognize loops but defer it to future work. A purely static analysis of the code is used in [24]. An analysis of traces (only dynamic analysis) without recovery of conditions or loops is used in [25]. In [26] the approach is to observe network communications. Reiss and Renieris compacting technique [27] is then used to aggregate several scenarios thereby recognizing loops and alternatives. This technique complements ours and is similar to our JAVA/RMI-specific approach [2]: they look at the boundaries of interacting applications while we look at the inside of the interacting applications.

In [28] the authors discuss the issues of reverse engineering FORTRAN legacy code, first being transformed into Java code, though without providing technical details regarding the reverse engineering of sequence diagrams. Cleve and Hainaut [29] conduct dynamic analysis of SQL statements for Data-Intensive applications.

A number of researchers have reported on ways to compact traces or sequence diagrams generated from them by either looking at the trace only (dynamic analysis only) or combining static and dynamic analysis [10, 16, 27, 30, 31]. The objective is for instance to recognize repeated (sub-)sequences of calls/messages and therefore loops. These works assume traces (or sequence diagrams) already exist. Instead we work on the generation of such traces (or sequence diagrams), attempting to minimize the overhead in terms of instrumentation and execution. These works are therefore complementary to our approach. Studying to what extent they can be combined is part

[1] References number 19, 21, 22, 23, 27, 29, 30, 33, 40, 90, 103, 116, 121, 123, 126, 129, 135, 141, 147 in [13].

of our future work. Other researchers suggest ways to dig into large sequence diagrams [10, 32, 33].

Many (commercial) tools are capable of reverse engineering sequence diagrams. (We omit the tools that limit the reverse engineering to the class diagram, such as Topcased, Poseidon, ModelMaker, Together, or MoDisco.) They either rely on a purely static analysis of the source code (e.g., MagicDraw, Rational Software Architect), or trace method executions/calls without collecting control flow information (e.g. MaintainJ, reverseJava, JSonde, javaCallTracer, J2U, TPTP's UML2 trace interaction View, CodeLogic). Note that Fujaba and related projects do not reverse engineer sequence diagrams. Some Fujaba projects do manipulate traces though, but for the purpose of detecting design patterns. With respect to tool support for reverse-engineering sequence diagrams, some authors discuss the right features such a tool should provide, especially when dealing with large traces/diagrams [33].

To conclude, with one exception (see below), no sequence diagram reverse engineering technique that we are aware of specifically addresses the issue of reducing the amount of collected runtime information and compensating this lack of information with a static analysis, with the attempt to limit the probe effect while still being able to show control flow information in sequence (scenario) diagrams. To the best of our knowledge, the approach we present in this paper, based on a combination of static analysis and dynamic analysis, is unique. Note however that our dynamic analysis, whereby we trace method calls, is not unique: in fact this appears to be the most widely used trace collecting technique. What is unique is our combination of static and dynamic analyses to provide more information in the generated sequence/scenario diagram.

As mentioned previously, Myers et al. [10] use both static and dynamic information to reverse engineer sequence diagrams. They collect the same information as we do in this paper (sections III to V), specifically line number and signature of invocations (static information) and invocations objects make to one another (dynamic information). They however rely on debug and source code analysis (static information) and byte-code instrumentation (dynamic information), while we only rely on source code analysis (for the static information) and bytecode instrumentation (for the dynamic information). Additionally, they have a different objective than ours: showing how static and dynamic information allows a tool to recognize loops; instead, we are interested in studying how combining both techniques reduces instrumentation, while also recognizing control flows.

Since we build on our previous work [1-3], it is worth discussing some of its details here. In [1-3] we used aspects (AspectJ [34]) to trace method entry and exit (around advice), conditions, loops, distributed information (focusing on RMI), and concurrency information (thread communications). In order to trace control flow information we had to instrument the source code in addition to use aspects since AspectJ did not offer any mechanism (i.e., join point) to do that. This limitation, in particular, prompted the current work: avoiding the instrumentation of the course code (in addition to the use of aspects), while still obtaining the same amount of information in the generated sequence diagrams.

To summarize our past instrumentation strategy [1-3], the dynamic analysis involves two calls to the trace logger for every method execution and two additional calls for each control flow structure (i.e., condition or loop), and requires the instrumentation of the source code (not only the bytecode). We consider this a heavy instrumentation approach: the execution overhead is high. We believe there is room for improvements, specifically in relation to the way synchronous messages (i.e., calls) and control flow structures are intercepted. Our approach attempts to improve those aspects, leaving aside the reverse engineering of concurrent and (RMI) distributed communications for which our past work is deemed sufficient [1-3]: our simplified instrumentation strategy (this paper) and our strategy to capture RMI and thread interactions should be easy to combine.

III. OVERALL DESCRIPTION OF THE APPROACH

Our approach is summarized in Figure 1. We attempt to minimize instrumentation, using aspects (activity a1 in Figure 1) and execute the instrumented version of the software using test cases (activity a2). In parallel, we reverse-engineer the control flow graph of methods (activity a3). We then combine the trace and control flow information in a model transformation activity (a4) to generate scenario diagrams.

In this process, we created tool support to automate activities a1, a3 and a4. Activity a2 can be automated, for instance using a framework like JUnit.

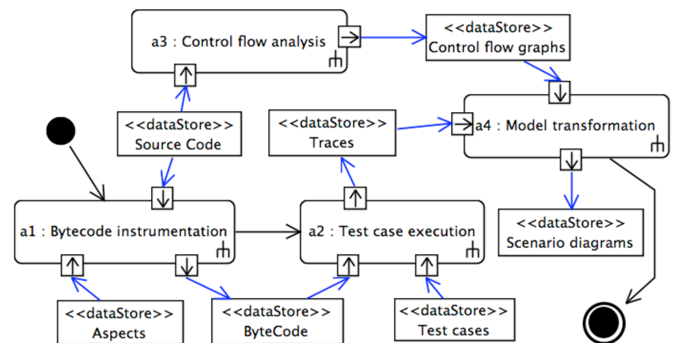

Figure 1 Proposed approach (UML activity diagram)

Our previous work has a similar process as the one of Figure 1. The main difference is the absence of activity a3 (and the generated graphs). The two approaches are equally easy to setup and use since the same activities are automated.

The remainder of the paper discusses the control flow and trace information (i.e., the models for `<<dataStores>> Control flow graphs` and `Traces` in Figure 1)—section IV, and our tooling to automate activities a1, a3, and a4—section V.

The main issues that drove the design of the trace model and the control flow model are:

- Collecting the right information in traces to reduce byte-code instrumentation to the minimum possible. The result is that we only collect method calls (identification of caller and callee objects, method signature, and line number of call—all in one trace log);

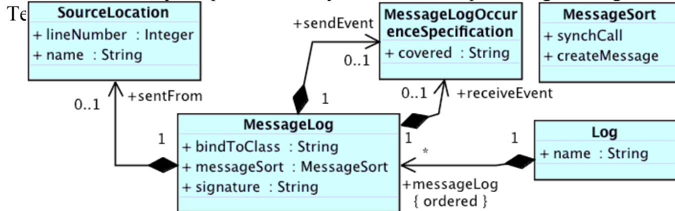

Figure 2. Trace model

- Being able to uniquely identify interacting objects: we devised a solution to this problem in our previous work, but had to revise it because we changed the logging strategy (specifically, we changed the aspects);
- Collecting the right information from the source code to match trace information, specifically line number of method calls, and to identify control flow structures: note that this analysis is performed offline;
- Devise models, especially the trace model, to facilitate model transformations: specifically, the trace model is close to the UML 2.1 metamodel.

IV. CONTROL FLOW AND TRACE INFORMATION

We refer the reader to [35] for the UML 2.1 metamodel and only discuss our trace model (section IV.A) and control flow model (section IV.B).

A. Trace model

The trace model (**Error! Reference source not found.**) presents execution trace data. It is designed to be very close in structure to the UML 2.1 Superstructure's `Message` components to facilitate transformations. In particular, the trace model's elements `Log`, `MessageLog`, `MessageLogOccurrenceSpecification` and `MessageSort` map to the UML's `Interaction`, `Message`, `MessageOccurrenceSpecification` and `MessageSort` respectively.

`Log` represents a single program execution and contains a sequence of `MessageLog`s. A `MessageLog` represents a message sent to the logger to signal the start of an execution, i.e., between a sending object and a receiving object (the two associations to `MessageLogOccurenceSpecification`). The `covered` attribute is a `String` containing the `ID` of an object (a unique identifier representing an object of a class), to be eventually translated into a lifeline in the sequence diagram. `MessageLog`'s attributes specify the kind of the message, i.e., a synchronous call or an object creation (`messageSort` maps to the UML's `Message`'s `messageSort`), the message's signature (in the form `returntype package.class.calledmethodname(arguments)`, i.e., the signature of the method being called), and the name of the class whose instance executes the called method (`bindToClass`). In the case of a static call, `bindToClass` contains the class defining this static method. This way, the transformation algorithm can determine the specific class and method invoked by the method call.

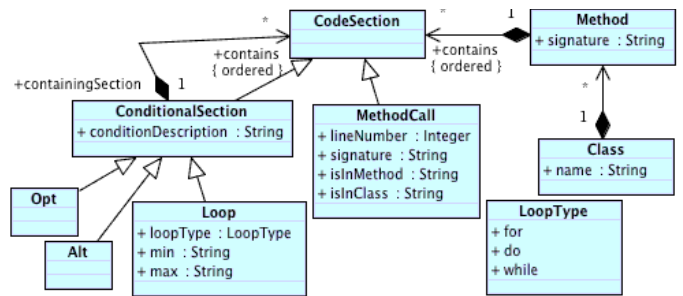

Figure 3. Control flow model

`MessageLog` contains a `SourceLocation`, which specifies the location (`name` of the class and `lineNumber`) in the source code from where the logged method call has been made. This, along with `bindToClass` allows us to match a `MessageLog` from the trace to the right element of the control flow model instance (i.e., the right methods in the caller and callee classes).

B. Control flow model

The control flow model (**Error! Reference source not found.**) captures a method's code structure in terms of method calls, possibly performed under conditions (alternatives, loops). It allows us to accurately locate method calls from the source code based on matching `MessageLog`s from the trace model and then place them into the UML sequence diagram structure. Knowledge of a method call's host method and, if the method call is inside a condition, its control flow structures, will allow us to accurately construct the sequence of executions with minimal dynamic (trace) data.

A `Class` whose behaviour is monitored contains `Methods` which in turn contain sequences of `CodeSections`. A `CodeSection` can be a `MethodCall` (we do not distinguish constructors) or a `ConditionalSection`, possibly nested (a `ConditionalSection` contains a sequence of `CodeSections`). A `ConditionalSection` is either an `Opt`, an `Alt` or a `Loop`. A `Loop` has a `LoopType` set to either `for`, `do` or `while`. Attribute `conditionDescription` (class `ConditionalSection`) specifies the actual condition. A `MethodCall` has a `lineNumber` from where it lies in the source file. `isInMethod` (class `MethodCall`) contains the signature of the method this method call is in. For `MethodCalls` outside a `ConditionalSection`, `method.signature` (navigating the association between `MethodCall` and `Method`, inherited from `CodeSection`, and accessing attribute `signature` of the calling method) is the same as `isInMethod`. However, `MethodCalls` inside a `ConditionalSection` do not have direct access to this association and therefore need to carry the `isInMethod` attribute. Attribute `isInClass` contains the name of the `Class` the `MethodCall` is in, and is needed for similar reasons as `isInMethod`.

An `Alt ConditionalSection` does not contain information about true/false branches because they are not handled as such in this work. The distinction between `Opt` and `Alt` is used in the mapping algorithm described in section V.C, hence they are kept as separate entities here.

V. TOOLING

We discuss below the aspects we used to collect runtime information (traces)—section V.A, the control flow model construction—section V.B, and the model transformation—section V.C. We only highlight the main principles due to space constraints. More details are available in Appendices (sections VIII and IX). An example illustrating the models and the model transformation is discussed in section V.D.

A. *Aspects*

The premise of this work is to provide a lighter instrumentation strategy than our previous work [1-3]. We therefore want to (1) avoid instrumenting control flow structures in the source code and (2) limit the impact of aspects, i.e., reduce the number of instrumentation calls made during execution.

To avoid instrumenting the source code, since AspectJ still did not provide pointcuts for control flow structures, we turned to static analysis, specifically a control flow graph created by parsing the source code. Note that even if AspectJ were providing such pointcuts, combining a static analysis with a dynamic analysis would still be preferable, as this would limit the probe effect (fewer aspects and pointcuts would be needed).

When combining the two kinds of analysis we need a way to match static information to dynamic trace. For example, we can do this by using the class name and method signature of executing methods as collected from the trace. This is not sufficient since we then do not know from where the call has been performed. Instead of instrumenting method executions, we therefore instrument method calls and capture the line number and the source file name from where the method call was made. Combined with a unique identification of executing objects, this information will allow us to correctly link dynamic and static data since this information (i.e., line numbers, method and class names) is also in the control flow graphs.

Once we can associate a method call from the trace to the location in the source code where that call is made (control flow graph), the static analysis allows us to determine from which method in which class the call was made and whether this call is inside a condition or a loop. Having obtained this location information statically, we no longer need to extract it through the trace as in [1-3].

Furthermore, when using a call joinpoint, AspectJ can provide information about both the source and destination methods. This allows us to further reduce the number of log statements compared to our previous work, an improvement we expect to translate into significantly lower overhead and faster execution time.

In the end, we have three aspects. The first one is to add to classes whose instances are monitored the capability to count and uniquely identify their instances. The second and third aspects intercept calls to methods and constructors and collect information before they are made: the join point is a call, the advice is a before advice. We selected call join point rather than execution join point since an execution join point is only aware of the location in the code of the method being called and not where the call is made from. Using a call join point, we can access information about the caller and the callee. The advice is a before advice rather than am after advice or an around advice since an after advice would lead to collecting messages in the reverse order, and an around advice is more expensive (in terms of instrumentation) than a before advice. The collected information includes: unique identifiers (or class name in case of static methods) of interacting objects, the signature of the method being called, and the line number where the call is made. The exact aspects can be found in Appendix VIII.

Transforming a trace into an instance of the trace model does not pose any technical difficulty and is not further discussed here.

B. *Control Flow*

We created a JavaCC (with JJTree) parser to generate instances of the control flow model, using a simplified Java grammar since we are only interested in class and method definitions, method and constructor calls, and control flow structures. Our parser can recognize class definitions, including inner class definitions, methods and method calls (including constructors), including method calls that are passed as parameters to other method calls and method calls that are inside condition statements; It handles if, else if and else, while loop, for loop (including for-each) but not ?: and the do-while loop (doing so is not a technical challenge). Note that when a condition contains a method call, the method call will appear in the control flow graph right ahead of the condition (outside of the conditional control flow construct). This is to better match the trace information, and the UML sequence diagram notation. Control flow generation is not further discussed here since it does not pose any technical difficulty.

C. *Model transformation*

We formalized the different steps of our transformation of instances of the trace and control flow models into an instance of the UML metamodel in terms of mapping rules between these models, using the Object Constraint Language (OCL) [11]. Creating an instance of the scenario diagram (using the UML sequence diagram notation) from instances of the trace and control flow models was then specified and performed with a third party, imperative model transformation tool named MD Workbench (http://www.mdworkbench.com/). The OCL rules can be seen as a specification for the MD Workbench transformation and were useful to identify whether our trace and control flow models had the required information to accurately perform transformations. The complete set of rules is not discussed here due to space constraints. Examples are available in Appendix IX.

D. *Illustrating example*

Let us now look at the example of Figure 4 to illustrate the two models and the essence of the model transformation. It has four parts: (I) an excerpt of code source showing the body of method `m()` in class `A`, which performs calls to methods `n()` and `m()` on an instance `b` that we assume to be of class `B`; (II) an instance of the control flow model (excerpt), i.e., the one of method `m()` in `A`; (III) a small excerpt of an instance of the control flow model, i.e., the one of class `B`; (IV) an (excerpt of

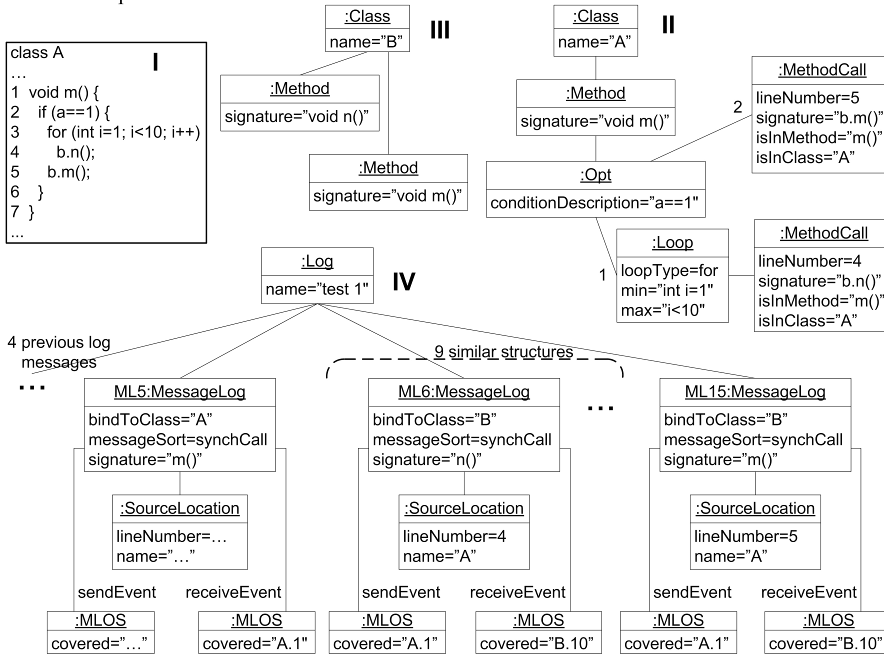

Figure 4 Illustrating example

an) instance of the trace model showing part of the execution of `m()` in `A`. The `lineNumber` attribute values in Figure 4, parts II and IV correspond to the line numbers in Figure 4, part I. In part IV, `MessageLogOccurenceSpecification` is simply referred to as `MLOS` for short. In Figure 4, part II, we recognize that `m()` contains an `Opt` alternative, which is itself made of a sequence of a `Loop` (performing a method call to `n()` on `b`) followed by a method call (to `m()` on `b`). The numbers `1` and `2` on the `Loop` and `MethodCall` sides of the links simply indicate that the links between the `Opt` object, and the `Loop` and `MethodCall` objects are ordered (as per the model in **Error! Reference source not found.**). In Figure 4, part IV, we assume that the execution of the program resulted in four initial log messages, followed by log messages `ML5`, `ML6`, …, `ML15`. Since the loop is executed nine times, there is a total of nine structures similar to `ML6` and its linked instances in the sequence (recall **Error! Reference source not found.**) of `MessageLog` instances linked to the `Log` instance. Instances of objects executing methods are uniquely identified thanks to our aspects, which is simply represented in part IV as strings `"A.1"` and `"B.10"`, suggesting for example that the instance of `A` executing `m()` is the first instance of `A` ever created in the program, and that calls to `n()` and `m()` are performed on the tenth instance of `B` created. In Figure 4, information not relevant to this discussion of the example is indicated with "…".

Let us now illustrate the essence of the model transformation. The excerpt of the trace model instance shows two instances: instance 1 of `A` and instance 10 of `B`. This allows the model transformation to create two lifelines; one for each of these instances. Instance `ML5` shows the call to `m()` on instance 1 of class `A`: the `receiveEvent MLOS` linked to `ML5`. The following `MessageLog` in the sequence from the `Log` instance, specifically `ML6`, shows a call to `n()` on the tenth instance of `B` (the `receiveEvent MLOS` linked to `ML6`) performed by the first instance of `A` (the `sendEvent MLOS` linked to `ML6`). Since there is no other `MessageLog` between `ML5` and `ML6`, and the target of the first `MessageLog` is the source of the second `MessageLog` (`A.1`), we can conclude that the call to `n()` in `ML6` is performed by `m()` which has been called in `ML5`. This allows the model transformation to create an execution specification on each life lines, showing the execution of `m()` on `A.1` and `n()` on `B.10`, as well as a message from the `m()` execution specification to the beginning of the `n()`'s execution specification. This also applies to the eight other `MessageLog` instances similar to `ML6`, as well as `ML15`. This results in the sequence diagram of Figure 5 (a).

The model transformation algorithm can also recognize that the call to `n()` on an instance of `B` recorded by `ML6` occurs at line 4 (attribute `lineNumber` of the `ML6`'s `SourceLocation` instance), that this call is performed in method `A.m()` (we already discovered that), and that, from the control flow model of method `A.m()`, the call to `n()` on an instance of `B` at line 4 happens in a loop, which itself happens in an alternative. Since this applies to all the `ML6` to `ML14` objects, the model transformation can collapse the nine messages labeled `n()` in Figure 5 (a) into one message in a `Loop` combined fragment, itself inside an `Opt` combined fragment. In addition, since `ML15` is a call that happens in `A.m()` at line 5 and that this call happens (control flow model) in the alternative, after the loop, we can then obtain the sequence diagram of Figure 5 (b).

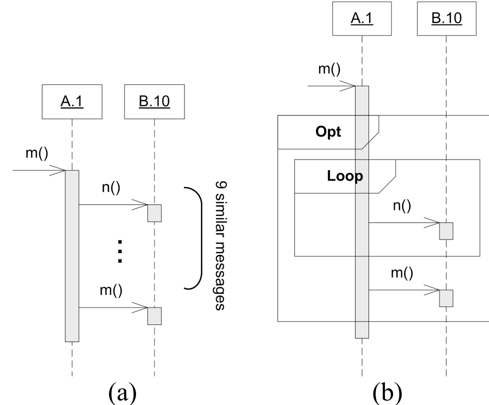

Figure 5 Illustrating the model transformation

VI. Case Study

We performed a case study with three research questions in mind: Is the execution overhead, measured as execution time, reduced when our approach is used compared to our previous work? Are the resulting scenario diagrams correct? Are they equivalent (we used UML 1.x and now use UML 2.x) to those produced by our previous technique?

A. Experiment set up

To answer these questions, we relied on four different case study systems (Table 1): The first one is a running Example we specifically built to exercise and control many different situations (e.g., nested loops and numerous iterations of loops). Through parameter values we can for instance control the amount of times method calls are performed, loops are executed, thereby simulating larger program executions. The code does not contain any computation, any GUI, any interaction with IO devices (e.g., reading a file). This should allow us to evaluate to what extent our new technique reduces the overhead as the size of the software (simulated by increasing the number of loops and calls) increases, without actually using a software larger than the four we detail in Table 1. Note however, that we expect execution overhead results to be worse with this example than with a real system exercising a similar pattern of calls and control flows, since we only have calls and control flow (e.g., no computation). In other words, the percentage of increased execution time would be smaller than what we report with this example; The second case study system is a software, developed by the second author in the context of a graduate course, implementing the Proof Carrying Code (PCC) technique [36], a technique for safe execution of untrusted code; The third system is a simple calculator, partly generated by JavaCC that heavily relies on the Visitor design pattern. We expect to see that pattern in the generated scenario diagrams; Last we used the Library system (server side only) from our previous work [1-3].

Our reverse engineering technique necessarily needs executions, i.e., test cases. Each case study was first executed with one test case that we selected to not produce too large traces, since manipulating large traces is out of the scope of this paper. We however used the same executions of the Library system as in our previous work [1-3], to allow comparisons. The Example system was also executed with varying parameter values to trigger large amounts of method calls and loop executions. We can then study the probe effect of our instrumentation on execution time.

To analyze execution overhead (question 1), using the three first systems (Table 1), we relied on three versions of each system: one with no instrumentation (base), one with our (light) instrumentation, and one with our (original) instrumentation. Note that to avoid a bias in favour of the light instrumentation, we removed the recording of node IDs and timestamps from the original instrumentation, necessary in the original technique to trace RMI and thread communications, thereby making the two instrumentation techniques comparable.

To compare execution times, we executed test cases 100 times twice at two separate occasions. Each test case for the three first systems is therefore executed 200 times. This is to control for the possible impact of the Windows operating system. (When possible, all other applications and services running on the computer were turned off, and the network was disconnected.) Between calls to the program under study, there is a call to a timer to get start and end times of the execution (in milliseconds). Note that this measure of time includes the start of the JVM. However, since we intend to compare execution times for the three versions, this should not have any impact on our conclusions.

Table 1. Characteristics of the four case study systems

Case study name	Classes	Methods	LOC	Question(s)
Example	4	13	56	1
PPC Prover	8	59	1280	1
Calculator	16	130	1175	1, 2
Library (Server)	44	459	3280	2, 3

To facilitate comparisons in the case of the third question, the test case used for the Library system is the one we used in the past [1-3]. When comparing the sequence diagrams generated by our two (original and light) techniques, we made abstraction of the facts that the original diagram abstracts out RMI and thread communication details and uses UML 1.x (instead of UML 2.x).

B. Results—Correctness of generated diagram

After investigations and comparisons with the source code, and available or expected diagrams (i.e., previous study using the Library system, behaviour of the visitor design pattern), we can conclude that the diagrams generated are accurate and provide as much information as our previous approach [1-3].

To illustrate this, we selected the sequence diagram for the `AddCopy` use case of the Library system, as available from the design documentation of the Library system: Figure 6. Figure 7 shows the reverse-engineered diagram for that behaviour, using our previous technique [1-3]. This illustrates the usefulness of the approach since for instance discrepancies with Figure 6 are clearly visible [1-3]. Some discrepancies are only due to the fact that we reverse engineer one scenario instead of complete sequence diagrams: e.g., not having in Figure 7 counterparts for messages in the `alt` combined fragment in Figure 6 (in the executed scenario there is no reservation on the title whose copy is added). Other discrepancies pertain to parameter types (e.g., `addCopy()`, `Copy`'s constructor). This example illustrates how using reverse engineered scenario diagrams can inform us about implementation choices. See [1-3] for a more detailed discussion.

Using the new approach we discussed in this paper, we obtain Figure 8. Note that we processed only the trace file of the server part of the Library System, explaining why the first lifeline of Figure 7 is not in Figure 8, and removed trace details such as server initialization and user logging to obtain comparable diagrams. The trace was processed by our prototype tool and the UML XMI file produced.

While looking at both Figure 7 and Figure 8, one can see that the same information is displayed: same messages, same ordering of messages. Two main differences are: we used UML 2.x, which allowed us to specify a combined fragment, whereas only UML 1.x was available at the time we created Figure 7; The message and lifeline labels have a different format in Figure 7 and Figure 8. These are stylistic differences due to our algorithm: i.e., the same information is available in the model. Furthermore, our algorithm does not handle RMI, so notions such as `Node 0` and `Node 1` (Figure 7) are not applicable.

Another example illustrating that the expected scenario diagram is generated can be found in the Appendix (section X):

we executed a simple calculator that implements the visitor design pattern and clearly observed object interactions dues to the pattern in the reverse-engineered scenario diagram.

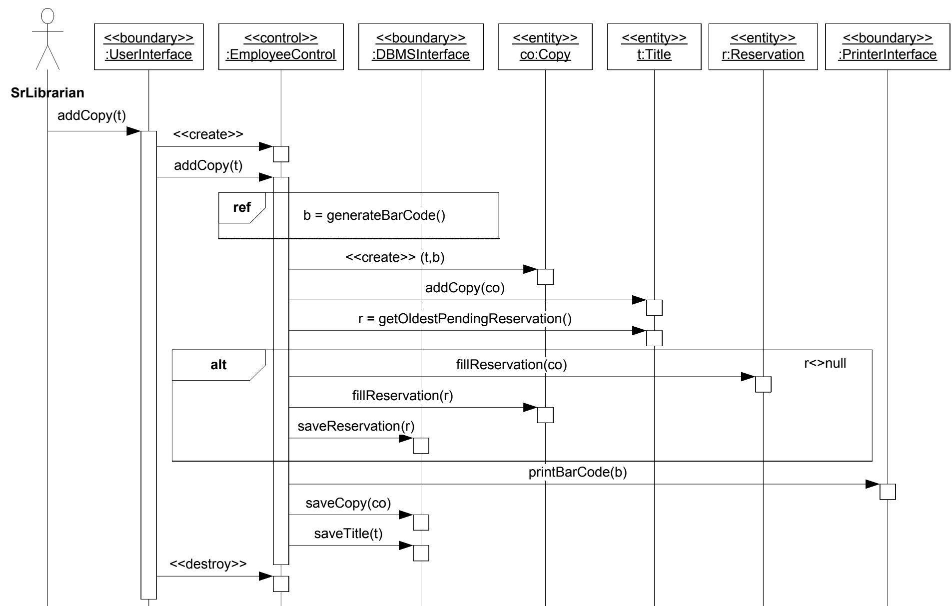

Figure 6 AddCopy sequence diagram from design document

C. Results--Overhead

Table 2 reports on the number of calls executed in the three versions of the three first case study systems. The first column reports on the calls executed in the non-instrumented versions whereas the other two columns indicate the overhead, i.e., additional calls (in fact calls to the logger), due to instrumentation strategies: e.g., tests executed 113 calls of the non-instrumented Calculator, whereas 131 additional calls are executed in the light instrumentation (for a total of 244 calls). Variations from system to system are due to differences in instrumentation techniques (e.g., we used an around advice in our previous work and now use a less demanding before advice) and the characteristics of the systems (e.g., amount of loops or if statements, amounts of calls to monitor). A general trend is that the original instrumentation is at least twice more expensive than the light one in terms of added calls. The differences between light and original is due to the fact that for the light instrumentation, the count is the sum of the number of method and constructor calls whereas for the original instrumentation, the count is the sum of twice (because of the around advice) the number of method and constructor calls and twice the number of conditions and loops encountered. Therefore, removing instrumentation of control flow structures helped reduce the probe effect, but our new aspects (before advice) also helped.

Table 2. Method calls counts

	No instrumentation	"Light"	"Original"
Calculator	113	131	305
PCC Prover	1138	1277	2334
Example	10000	10003	32000

Execution times are reported in Figure 10: box plots indicate minimal and maximal execution time values (opposing ends of the vertical line) as well as first and third quartile lines encompassing time range achieved by half of the total executions. While we tried to control other operating system activities, there is still a large variation in the execution times obtained. However, compared to the differences between minimum and maximum execution times, most execution times lie within a narrow range (so we can discard the outliers). Notice that the largest number of points for each variation is found at or near the minimum execution time. This is likely due to the fact that most of the time there were very few other processes running on the computer. The higher points probably occurred during times that the processor was handling other expensive system events we were not able to control. On average, the light instrumentation approach causes the program to execute slower than it would without instrumentation but much faster than with the original instrumentation approach, especially as the number of method calls grows, i.e., the size of the instrumented program grows: Example requires more calls than PCC Prover, which requires more calls than Calculator (Table 2).

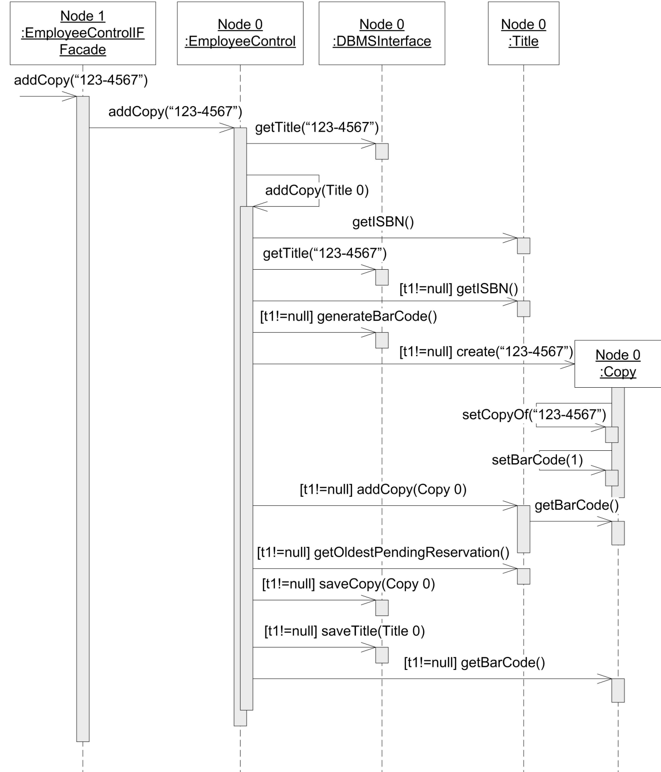

Figure 7 AddCopy scenario diagram from [1-3]

We compared the samples of execution times obtained with the Light and Original versions and with the Original and Base versions, for all three systems: we used a one-tailed t-test and confirmed the results with the corresponding non-parametric

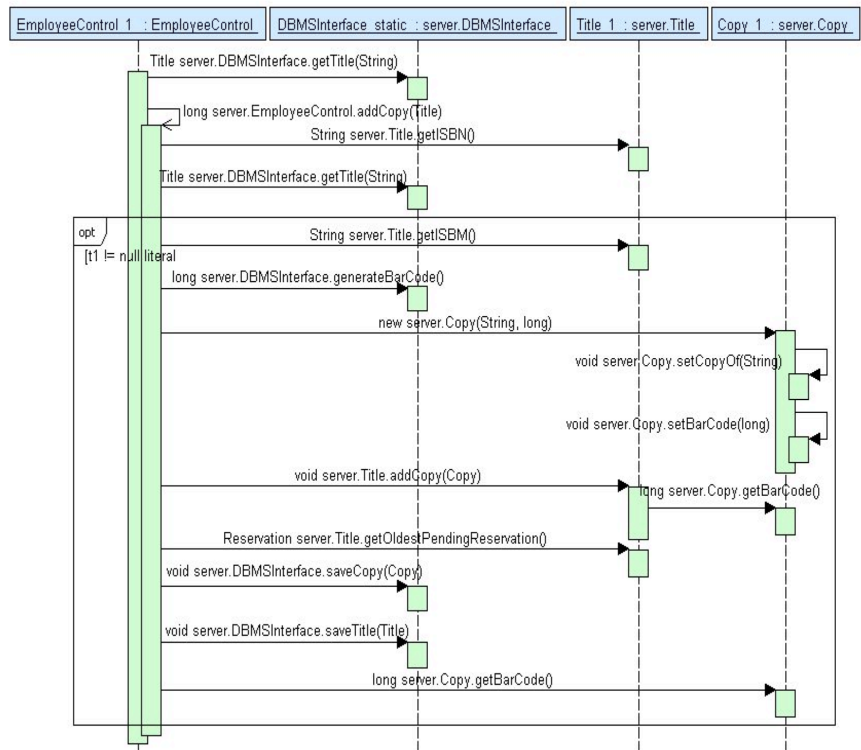

Figure 8 AddCopy scenario diagram generated by the new technique

Wilcoxon signed-rank test. All comparisons are statistically significant (p-value threshold at 0.05), with all the p-values smaller than 0.0001, except one (p-value=0.004) obtained when comparing Light and Original for Calculator. The light instrumentation statistically leads to less overhead than the original instrumentation.

The variation between the fastest time and the median is noticeable for Calculator light, PCC Prover original and Example original. We attribute this to the instrumentation making system calls to write to a log file. Because these calls are external to the java execution, they may be more susceptible to external processes. A file needs to be created and written to repeatedly, which are execution-heavy tasks (much heavier than any instrumentation-related behaviour added to the programs). We will investigate other logging mechanisms than writing to a file in the future. For Calculator, the light version is more negatively affected by the instrumentation than the original despite a lower number of instrumentation calls. This could be because the amount of characters written into a file for our light instrumentation is higher than for original. For example, the first trace entry for Calculator light was 249 characters long, versus 175 for original. Light instrumentation is therefore not "lighter" than original for executions small enough not to be negatively affected by large number of instrumentation calls. Indeed, even though the original instrumentation writes to a file at least twice as often as our light instrumentation, the number of times the file is written to is small so the difference is not important. We suspect that the overhead when the program is small mostly comes from creation of the trace file.

We also simulated the instrumentation impact on execution time for larger systems thanks to Example, which allows us to control the number of method calls being executed thanks to an input argument. We executed 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, and 10^8 calls, 100 times each. Figure 9 shows the results, using a logarithmic scale. Again, the light instrumentation approach causes the program to execute slower than it would without instrumentation but much faster than with the original instrumentation approach: data show that the light instrumentation is 2 times faster than the light one for 10^5 executions and above. The figure also shows that additional work is required to further reduce the impact of instrumentation: a different tracing mechanism can be used as we have already mentioned; one may also consider instrumenting only parts of a large program to reverse engineer.

VII. CONCLUSION

In this paper, we built on our previous work towards the automated generation of scenario diagrams by reverse engineering the source code. Our objective was to combine a dynamic analysis of program executions (traces) with a static analysis of the source code to (1) obtain scenario diagrams that are equivalent to what our previous technique can generate (i.e., sequences of messages with information on conditions and loops triggering those messages, represented under the form of the UML sequence diagram), while (2) reducing the amount of instrumentation of the bytecode and avoiding instrumenting the source code. The latter is particularly important as we do not want the instrumentation to affect the program behaviour, with the risk of not observing the right behaviour when executing the instrumented program.

We therefore first tried to reduce the impact of our aspects for trace generation (e.g., we trace calls rather than executions, with a before advice rather than an around advice). In parallel we generated control flow graphs of the methods to be instrumented: we were only interested in method definitions, the method calls they trigger and the conditions under which those calls are triggered. We represented both sets of information using UML class diagrams. Generating a UML scenario diagram then became a model transformation problem from an instance of the trace model and instances of control flow models to an instance of the UML metamodel.

We performed several case studies that indicate that we achieved our goals: (1) the generated diagrams were equivalent to the ones generated by our previous technique; (2) we reduced the probe effect due to instrumentation.

There is room for future work. First, we intend to combine our new instrumentation strategy with our past technique to monitor RMI and thread communications. Second, our experimental results show we can still reduce execution time overhead by for instance considering other mechanisms than a file to collect trace information. We also intend to perform more extensive experimentations to more precisely understand what aspects of the approach hurts the most in terms of probe

effect, given characteristics of the program being monitored. We also intend to combine our technique with existing trace minimization techniques (e.g., [10, 16, 27, 30, 31]). Last, the next challenge will be to combine several scenario diagrams and create accurate, complete sequence diagrams. There is work in the literature we can get ideas from [13].

REFERENCES

[1] L. C. Briand, Y. Labiche, and J. Leduc, "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE TSE,* vol. 32, pp. 642-663, 2006.
[2] L. C. Briand, Y. Labiche, and J. Leduc, "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multithreaded Java software," Carleton University, TR SCE-04-04, September 2004.
[3] L. C. Briand, Y. Labiche, and J. Leduc, "Tracing Distributed Systems Executions Using AspectJ," in *IEEE ICSM*, 2005, pp. 81-90.
[4] T. Pender, *UML Bible*: Wiley, 2003.
[5] D.-P. Nguyen, C.-T. Luu, A.-H. Truong, and N. Radics, "Verifying Implementation of UML Sequence Diagrams Using Java PathFinder," in *Knowledge and Systems Engineering*, 2010, pp. 194-200.
[6] Z. Zhou, L. Wang, Z. Cui, X. Chen, and J. Zhao, "Jasmine: A Tool for Model-Driven Runtime Verification with UML Behavioral Models," in *IEEE High Assurance Systems Engineering*, 2008, pp. 487-490.
[7] Y.-G. Guéhéneuc, "A reverse engineering tool for precise class diagrams," in *conference of the Centre for Advanced Studies on Collaborative research*, 2004, pp. 28-41.
[8] B. Beizer, *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, 1990.
[9] Y.-G. Guéhéneuc and T. Ziadi, "Automated Reverse-engineering of UML v2.0 Dynamic Models," in *ECOOP Workshop on Object-Oriented Reengineering*, 2005.
[10] D. Myers, M.-A. Storey, and M. Salois, "Utilizing Debug Information to Compact Loops in Large Program Traces," in *CSMR*, 2010, pp. 41-50.
[11] J. Warmer and A. Kleppe, *The Object Constraint Language*, 2nd ed.: Addison Wesley, 2003.
[12] B. Kolbah, "Reverse Engineering of Java Programs through Static and Dynamic Analysis to generate Scenario Diagrams," Masters of Applied Science Thesis, Carleton University, Ottawa, January, 2011.
[13] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE TSE,* vol. 35, pp. 684-702, 2009.
[14] B. A. Malloy and J. F. Power, "Exploiting UML Dynamic Object Modeling for the Visualization of C++ Programs," in *ACM Symposium on Software Visualization*, 2005, pp. 105-114.
[15] T. Systa, K. Koskimies, and H. Muller, "Shimba - An Environment for Reverse Engineering Java Software Systems," *SPE,* vol. 31, pp. 371-394, 2001.
[16] A. Hamou-Lhadj and T. C. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System," in *IEEE ICPC*, 2006, pp. 181-190.
[17] K. Koskimies, "Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs," in *IEEE ICSE*, 1996, pp. 366-375.
[18] J. Koskinen, M. Kettunen, and T. Systa, "Profile-Based Approach to Support Comprehension of Software Behavior," in *ICPC*, 2006, pp. 212-224.
[19] F. Q. Wang, H. J. Ke, and J. B. Liu, "Towards the Reverse Engineer of UML2.0 Sequence Diagram for Procedure Blueprint," in *World Congress on Software Engineering*, 2009, pp. 118-122.
[20] S. Munakata, T. Ishio, and K. Inoue, "OGAN: visualizing object interaction scenarios based on dynamic interaction context," in *IEEE ICPC*, 2009, pp. 283-284.
[21] T. Ishio, Y. Watanabe, and K. Inoue, "AMIDA: a sequence diagram extraction toolkit supporting automatic phase detection," in *Companion of the 30th ICSE*, 2008.
[22] O. Pilskalns, S. Wallace, and F. Ilas, "Runtime debugging using reverse-engineered UML," in *Models*, 2007, pp. 605-619.
[23] M. H. Alalfi, J. R. Cordy, and D. Thomas, "Automated reverse engineering of UML sequence diagrams for dynamic web applications," in *IEEE ICST Workshops*, 2009, pp. 287-294.

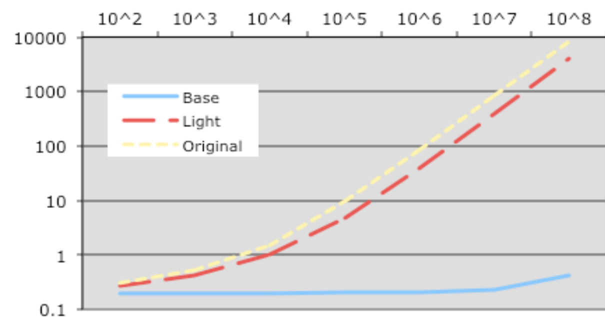

Figure 9. Median execution times while increasing the number of method calls

[24] Y. Imazeki and S. Takada, "Reverse engineering of sequence diagrams from framework based web applications," in *IASTED International Conference on Soft. Engineering and Applications*, 2009, pp. 202-209.
[25] A. Serebrenik, S. Roubtsov, E. Roubtsova, and M. van den Brand, "Reverse engineering sequence diagrams for Enterprise JavaBeans with business method interceptors," in *WCRE*, 2009, pp. 269-273.
[26] C. Ackermann, M. Lindvall, and R. Cleaveland, "Recovering views of inter-system interaction behaviors," in *WCRE*, 2009, pp. 53-61.
[27] S. P. Reiss and M. Renieris, "Encoding Program Executions," in *ICSE*, 2001, pp. 221-230.
[28] S. R. H. Hoole and T. Arudchelvam, "Reverse engineering as a means of improving and adapting legacy finite element code," in *International Conference on Industrial and Information Systems*, 2009, pp. 227-232.
[29] A. Cleve and J.-L. Hainaut, "Dynamic Analysis of SQL Statements for Data-Intensive Applications Reverse Engineering," in *WCRE*, 2008.
[30] A. Hamou-Lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension," Ph.D., University of Ottawa, 2006.
[31] P. Dugerdil and J. Repond, "Automatic generation of abstract views for legacy software comprehension," in *ACM India software engineering conference*, 2010, pp. 23--32.
[32] K. Noda, T. Kobayashi, K. Agusa, and S. Yamamoto, "Sequence Diagram Slicing," in *Asia-Pacific Soft. Eng. Conference*, 2009, pp. 291-298.
[33] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," *Software Maintenance and Evolution,* vol. 20, pp. 291-315, 2008.
[34] J. D. Gradecki and N. Lesiecki, *Mastering AspectJ - Aspect-Oriented Programming in Java*: Wiley, 2003.
[35] OMG, "UML 2.0 Superstructure Specification," Object Management Group, Final Adopted Specification ptc/2007-11-02, 2007.
[36] G. C. Necula, "Proof-carrying code," in *ACM SIGPLAN-SIGACT Symposium on Principles of Prog. Languages*, 1997, pp. 106--119.

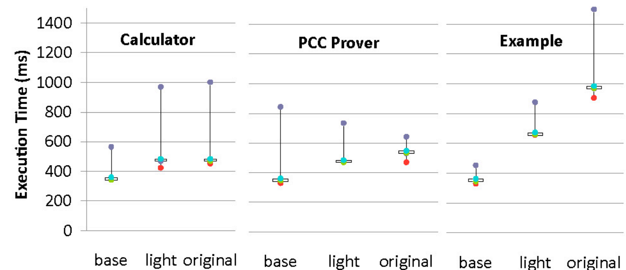

Figure 10. Execution times for different instrumentations

VIII. Appendix—Aspects

A. Logger

Class Logger implements the logging functionality and is a simplified version of the one we used in [1-3] since tracing multithreading and RMI is outside of scope for this paper. Specifically, unlike our previous Logger, we do not need to retain a timestamp of the trace statement, node on the network or thread identifications.

The Logger implements the singleton design pattern. All trace statements generated by our aspects (see following sections) are written into a single file, Trace.txt by calling method instrument(List <String> record) of the Logger. The Logger needs to be generated for every project to be instrumented but this can be done automatically.

B. Object identification

Interacting objects need to be uniquely identified to draw lifelines in the generated sequence diagram. An aspect (Figure 11) and the ObjectID interface are used to correctly set and make available to other aspects a unique identifier and class name for objects whose behavior is to be monitored/traced. The ObjectID interface simply specifies one method called getObjectID() that returns an integer: a unique identifier for the instance of the class on which it is called. Two instances of the same class cannot have the same identifier (this is ensured by the aspect), while instances of different classes can have the same identifier. Instances of a class are therefore uniquely identified using their identifier, while instances in the system, possibly from different classes, are uniquely identified using the unique pair (class name, identifier). Setting an object's identifier happens during construction of the object.

The aspect in Figure 11 is specific to one class called MyClass, to make the discussion more concrete and easier to follow. Creating a similar aspect for each class to be monitored at runtime is straightforward. This aspect adds an attribute of type int, named objectID to the instrumented class (line 1). It also specifies that the MyClass class implements a new interface, specifically ObjectID (line 13), and adds an implementation to the method declared in this interface, specifically getObjectID() (lines 14-19).

The aspect also adds to the class (static attribute currentObjectID and method objectIDgenerator()) the capability to count its instances, which is the mechanism used to set a unique identifier to those instances. Attribute objectID is obtained for each object of the instrumented class during its creation by calling static method objectIDgenerator(objectID). This method initializes attribute objectID by incrementing static attribute currentObjectID (line 7). At this time, a call is made to the Logger to record that this object has just been created (lines 8-9), since the first time this method is called is during construction. The Logger then records a lifeline to match the UML terminology: an object executing a method will eventually be represented as a lifeline in the sequence diagram. During the lifetime of this object, its unique ID, e.g., "MyClass_12" (lines 8, 18), is accessed by calling method getObjectID(), specified by the ObjectID interface.

If the object created is of a class that has parent classes and those classes have a behavior that is also intercepted, the objectIDGenerator method will be called for each parent and then for the child, following the order in which constructors in an inheritance hierarchy are called in Java. This will cause the Logger's instrument() method to be called for each of these constructors in sequence from parent to child. If the parent class has a constructor, it will be called before the child's objectID attribute is set by objectIDGenerator(). If then the parent's constructor contains method calls, the child's objectID could be read by the aspect before it is initialized to its final value, as specified in objectIDgenerator(). The trace file would then have incorrect information pertaining to the ID of the object: calls in the parent constructor to methods overridden in the child would lead to log entries with the wrong object identifier. To avoid this, whenever getObjectID() is called, a check is made whether the objectID attribute has been initialized (line 15). If the initialization has not happened, initialization is performed before getObjectID() returns (line 16). To avoid overwriting this value at actual initialization, the objectIDgenerator() verifies (line 5) whether the objectID has already been initialized and does not modify it if this is the case. Since objectIDgenerator() is static, it cannot access the objectID attribute of the instance. This information therefore has to be passed as a parameter: lines 16 and 1.

As a result, if classes A, B, and C are monitored, A being a parent for B and B being a parent for C, creating an instance of C results in executing line 8 of Figure 11 three times: once for A, once for B, and once for C, in that order. The trace will contain log statements like (class names would be fully qualified):

```
<lifeline className="A" name="A_1"/>
<lifeline className="B" name="B_1"/>
<lifeline className="C" name="C_1"/>
```

These indicate that instances of classes A, B, and C have been created and they all have unique ID 1 (within their class). Obviously, no instance of A and B has been created really. Those statements are simply a byproduct of our procedure to collect information. The two first statements can simply be

```
1 private int MyClass.objectID =
     MyClass.objectIDgenerator(objectID);
2 private static int MyClass.currentObjectID = 1;
3 private static int MyClass.objectIDgenerator(int i) {
4    int id = i;
5    if (i < 1){
6      LinkedList <String> log=new LinkedList <String> ();
7      id = MyClass.currentObjectID++;
8      log.add("<lifeline className=\"example.MyClass\"
          name=\"MyClass_" +  id + "\"/>");
9      Logger.getLoggingClient().instrument(log);
10   }
11   return id;
12 }
13 declare parents : MyClass implements ObjectID;
14 public String MyClass.getObjectID() {
15    if (objectID < 1){
16    objectID = MyClass.objectIDgenerator(objectID);
17    }
18    return "MyClass_" + objectID;
19 }
```

Figure 11. Identifier aspect for class MyClass

removed from the log file, in a post-processing phase without any impact on the accuracy of the result.

We investigated alternatives to avoid this (very simple) post-processing but they all resulted in additional instrumentation, which goes against our objective to limit the probe effect. The instrumentation of parents (to generate IDs) adds a slight overhead; however, we judged it to be small enough to use this approach until a significantly better one is available.

C. *Method/Constructor call interception*

Figure 12 shows our pointcuts to intercept method calls (either to instance or static methods). Pointcut `callMethod` specifies all method calls (line 1), i.e., with any method name and any list of parameters, except for static method calls (line 2) and calls made to the `IdentifierAspect`, the `Logger` or `ObjectID` classes/aspects (lines 3-5). The pointcut for intercepting calls to static methods is similar (line 6-10). Finally, pointcut `callConstructor` specifies calls to constructors (line 11), omitting calls to constructors in the instrumentation infrastructure (line 12), assumed to be in an `example` package (we opted for simplicity, a different package name can easily be used).

Advices for these pointcuts are all executed **before calls**, rather than **around executions** [1-3] as we want to know the location of calls in the source code to match intercepted calls with control flow information. An **execution** join point is only aware of the location in the code of the method being called and not where the call is made from. We can deduce the source code location of the executed method through static analysis by matching its signature, given that we know the type of the object executing the method, hence we do not need another join point that would provide such information. We use a **before** rather than an **after** advice because we want to obtain the sequence in which method calls are made: an after advice would provide the reverse order (e.g., log entry for the callee before a log entry for the caller).

The aspect code executed at the pointcut obtains the information specified earlier. Figure 13 presents the code, executed right before a `callMethod` pointcut, thereby obtaining information about a call to a method of an object, and sending that information to the `Logger`. Local variable `thisID` (line 2) records the ID of the object making the call. Line 3 defines a local variable to store log information before sending it to the `Logger` (line 19). Lines 4-9 retrieve the identity of the object performing the call: either an object

```
1 pointcut callMethod() : call (* *.*.*(..))
2           && !call (static * *.*.*(..))
3           && !call (* example.IdentifierAspect.*(..))
4           && !call (* example.Logger.*(..))
5           && !call (* example.ObjectID.*(..));


6 pointcut callStaticMethod() : call (static * *.*.*(..))
7           && !call (* example.MethodAspect.*(..))
8           && !call (* example.IdentifierAspect.*(..))
9           && !call (* *.*.objectIDgenerator(..))
10           && !call (* example.Logger.*(..));

11 pointcut callConstructor() : call (*.*.new(..))
12           && !call (example.Logger.new(..));
```

Figure 12. Pointcut definitions

whose behavior is instrumented and implements the `ObjectID` interface, and then we retrieve the object identifier (line 5), or the calling context is static (e.g., the main method of the program), or is a class that has not been instrumented (line 8). Class file names are formatted by `getStaticClassName(…)`.

Next, the advice retrieves the unique identifier of the object being called (line 11): `thisJoinPoint.getTarget()`; and stores it in local variable `targetID`. Then the aspect retrieves details on the method being called: class name (line 12, thanks to `getBindToClassName`), method signature (line 12, thanks to `getMethodSignature`), line number where the call is located (line 17, thanks to `getLineNumber`).

Currently the case of an instrumented class making a call to an object of an un-instrumented class is not supported. Modifying the aspect to handle such a case does not pose any technical difficulty.

The aspect for pointcut `callStaticMethod` is very similar and is therefore not shown here. The main difference is that we do not have any target object identifier to report on. Instead the trace indicates the name of the class defining the static method being called.

The aspect to report on calls to constructors, i.e., `callConstructor`, is also very similar. It starts with the identification of the identity of the object performing the call (lines 4-9 of Figure 13), and ends with the formatting of the trace message (similarly to lines 12-18 of Figure 13). There are two important differences though, due to the fact that the object creation has not yet finished at the point where the advice is executed (i.e., right before the call to the constructor). First, the target is not an object with an ID (yet): lines 10-11 of Figure 13 are therefore removed. Second, line 16 of Figure 13 cannot refer to `targetID`. Instead, we use a keyword, specifically `nothing`, in place of this ID.

As a result, assuming class `C` inherits from `B`, which inherits from `A`, and an instance of `C` is created, the log will contain the following statements:

```
<messageLog bindToClass="C"
   messageSort="createMessage"
   signature="new C(int)">
<sendEvent covered="C_static"/>
<receiveEvent covered="nothing"/>
<sentFrom lineNumber="6" name="C.java"/>
</messageLog>
<lifeline className="A" name="A_1"/>
<lifeline className="B" name="B_1"/>
<lifeline className="C" name="C_1"/>
```

A statement for any intercepted call performed by the constructors of `A`, `B`, and `C` would follow. As discussed earlier, the `<lifeline…` statements for classes `A` and `B` should be removed. Then, in a post-processing step, it becomes easy to recognize that the created object of class `C` in the first `<messageLog…` (so far identified as `nothing`) is in fact the object with ID `C_1` (from the last `<lifeline…` statement). We considered alternatives to avoid this very simple post-processing step, but they all resulted in additional instrumentation.

```
1 before(): callMethod () {
2   String thisID = new String ();
3   LinkedList <String> log = new LinkedList <String> ();
4   if (thisJoinPoint.getThis() != null) {
5      thisID = String.valueOf(((ObjectID)
                   thisJoinPoint.getThis()).getObjectID());
6   }
7   else {
8      thisID = getStaticClassName(thisJoinPointStaticPart.
                 getSourceLocation().toString());
9   }
10  String targetID = new String ();
11  targetID = String.valueOf(((ObjectID)
thisJoinPoint.getTarget()).getObjectID());
12  log.add("<messageLog bindToClass=\""
      + MethodAspect.getBindToClassName(
         thisJoinPoint.getTarget().toString())
      + "\" messageSort=\"synchCall\" signature=\""
      + MethodAspect.getMethodSignature(thisJoinPoint.toString())
      + "\">");
15  log.add("<sendEvent covered=\""+ thisID + "\"/>");
16  log.add("<receiveEvent covered=\""+ targetID + "\"/>");
17  log.add("<sentFrom lineNumber=\""
      + MethodAspect.getLineNumber(
         thisJoinPointStaticPart.getSourceLocation().toString())
      + "\" name=\"" + MethodAspect.getFileName(
         thisJoinPointStaticPart.getSourceLocation().toString())
      + "\"/>");
18  log.add("</messageLog>");
19  Logger.getLoggingClient().instrument(log);
20  }
```

Figure 13. Code executed before pointcut `callMethod`

The aspects for intercepting calls to instance methods, static methods and constructors are all generic and can be automatically generated from a list of classes whose behaviour needs to be monitored.

IX. Appendix—Model transformation rules

Mapping rules are defined in the context of a specific class, which we named `Matching`. This allows us to formally specify the rules without having to modify any of the three models (which are considered as data only): trace model, control flow model, and more so the UML metamodel.

First we link each log of a method call (`MessageLog` in the trace model) to a method call in the source code (`MethodCall` in the control flow model), by matching their respective signatures, sending classes and line numbers.

```
Matching :: matchMessageLogToMethodCall(MessageLog
   ml) : MethodCall
post :
  result=MethodCall->allInstances->select(mc|
    ml.signature=mc.signature
    and ml.sentFrom.name=mc.isInClass
    and ml.sentFrom.lineNumber=mc.lineNumber )
```

Next, there are elements in the sequence diagram that are specific to every `MessageLog` of the trace model and are not shared with another `MessageLog`'s elements. If `MessageLog` is of sort `createMessage` or `synchCall`, these unique elements are two `Message`s (one for the call and one for the return), four `MessageOccurrenceSpecification`s (two for the call message and two for the return message), two `SendOperationEvent`s, two `ReceiveOperationEvent`s and a `BehaviourExecutionSpecification`. Other elements related to the `MessageLog` may be shared with other `MessageLog`s and are created for a `MessageLog` only if the matching ones do not yet exist in the model. These elements include: one or two `Lifeline`s, and the `Property` representing these `Lifeline`s, an `Actor` for the initial `Lifeline` corresponding to the first `MessageLog` in the trace and `Class`es for the remaining `Lifelines`, `Connector`s, and `ConnectorEnd`s, `Operation`s as well as elements representing control flow, namely: `CombinedFragment`, `InteractionOperand`, `InteractionConstraint`, and `OpaqueExpression`.

For instance, the OCL expression below characterizes the unique elements of messages that are derived from a log, using operations `mapSendMessage()`, `mapReplyMessage()`, and `mapToBES()` which, in a nutshell, identify whether the log message sort is the UML message sort, the UML message connectors belong to lifelines that correspond to the caller and callee identified in the log (class name, object identifier), the UML message has the right comment attached to it. For each message log `ml` (line 1), there must be a message `m_s` (line 5) in interaction `i` (line 3) and a return message `m_r` (line 7) whose characteristics (e.g., caller and callee objects) map `ml`'s characteristics. Lines 10-11 also map `ml`'s characteristics to a behaviour execution specification in interaction `i`.

```
1  Matching::SDElementsForMessageLog(MessageLog ml)
2  post :
3    Interaction.allInstances->exists(i |
4      i.message->exists(m_s : Message |
5        Matching.mapSendMessage(m_s, ml))
6        and
7        i.message->exists(m_r : Message |
8          Matching.mapReplyMessage(m_r, ml))
9        and
10       i.fragment->select(oclIsKindOf
           (BehaviourExecutionSpecification)
11           ->exists(bes|Matching.mapToBES(bes, ml))
```

Combined fragments represent control flow structures that are enclosing messages (i.e., method calls). If a `methodCall` is inside a `conditionalSection`, then the corresponding `Message` in the sequence diagram will be inside a `CombinedFragment`. If the `conditionalSection` is nested inside another `conditionalSection`, then the resulting `CombinedFragment` will be inside another `CombinedFragment`.

More than one `MessageLog` can be associated with a `CombinedFragment`, i.e., a conditional section in the code can contain several method calls, so a new `CombinedFragment` will only be created once the algorithm determines that an existing combined fragment with appropriate settings does not already exist in the `Model`.

Calls made from a loop may execute widely different paths. Because in our work each "loop" combined fragment represents a single transversal of the loop, there will never be a case where if and else sections are found inside a single "alt" combined fragment as they would be in manually generated diagrams. Our approach does not yet recognize repeated iterations of a loop as one combined fragment. The problem of combining several scenario diagrams into one sequence diagram, thereby merging alternative and repeated executions (sub-traces) is deferred to future work. Instead, a new combined fragment is added to the sequence diagram for each iteration of a loop.

Although they are not necessary in a complete sequence diagram, our transformation also adds `Comment`s to `Message`s, `Operation`s and `CombinedFragment`s, indicating their relationship to the source code such as class name and line number where the related method calls are found.

X. Appendix—Scenario diagram correctness

To answer the question whether the scenario diagram obtained by our tool is correct, we also considered the scenario diagram resulting from the execution of a Calculator. The Calculator implements the Visitor design pattern, which should allow us to visually identify whether the generated diagrams are correct or not (the Visitor design pattern involves very specific object interactions). From the documentation and source code we know that the Calculator is an implementation of a Visitor pattern and that it traverses the math question (in our case “1+1”) in the form of an abstract syntax tree (AST).

We have obtained the trace by running the Calculator. Because for our purpose we are not interested in how the tree is built, but only how it interacts with the visitor during an evaluation (here, “1+1”), we deleted all trace statements prior to the `SumVisitor` constructor call from the trace file. The tool was run and the results displayed in Figure 14 and Figure 15.

The AST is clearly visible in the scenario diagram. `ASTExpression` (1+1) has one child, `ASTOperator` (+) who has two children, two `ASTOperands` (1). The visitor pattern is likewise clearly visible. After the `SumVisitor` object is created (first message in Figure 14), the `accept()` method is invoked on `ASTExpression` and the `Visitor` object is passed as parameter.

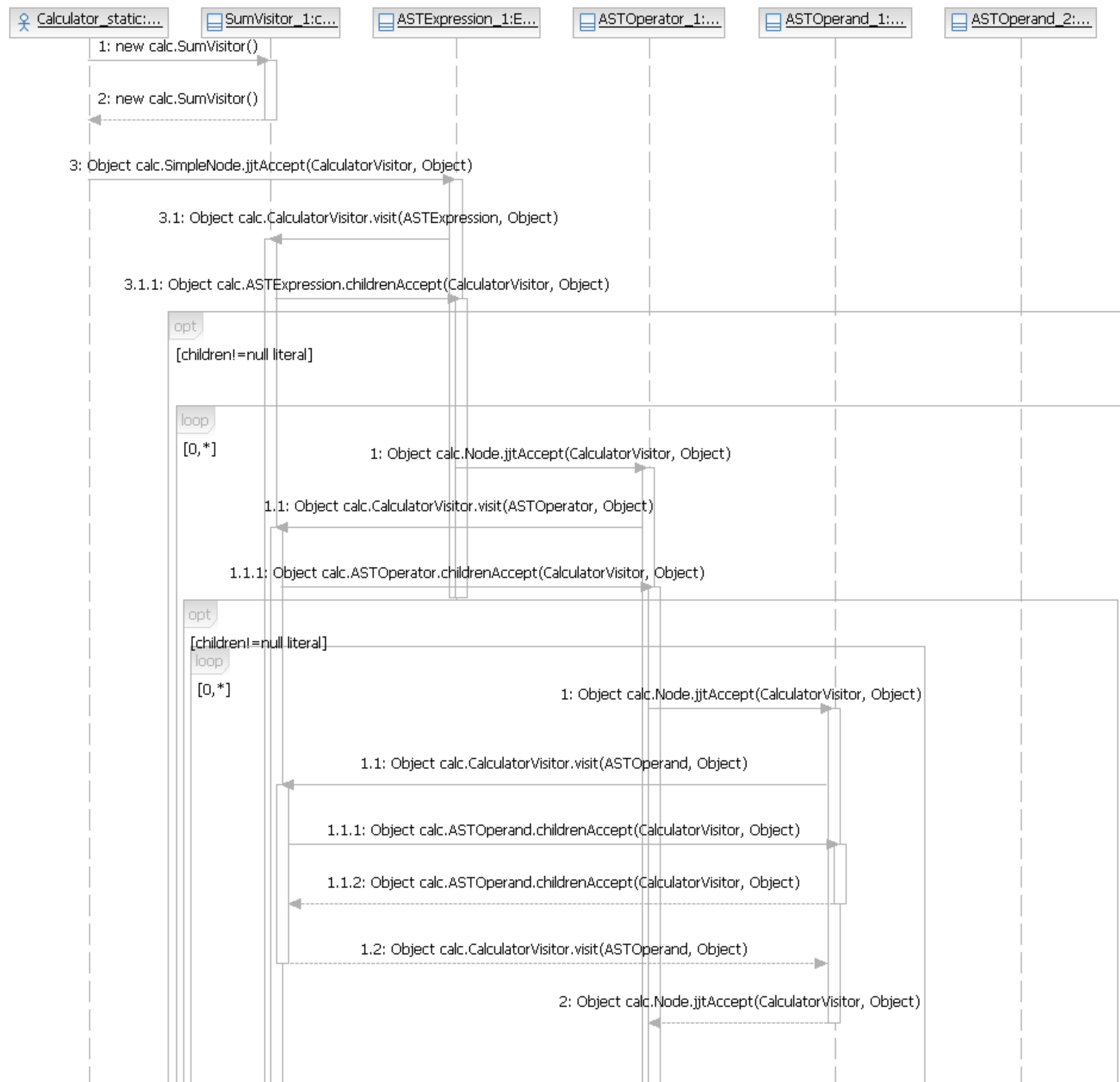

Figure 14 Reverse engineered scenario diagram for the Calculator (part I)

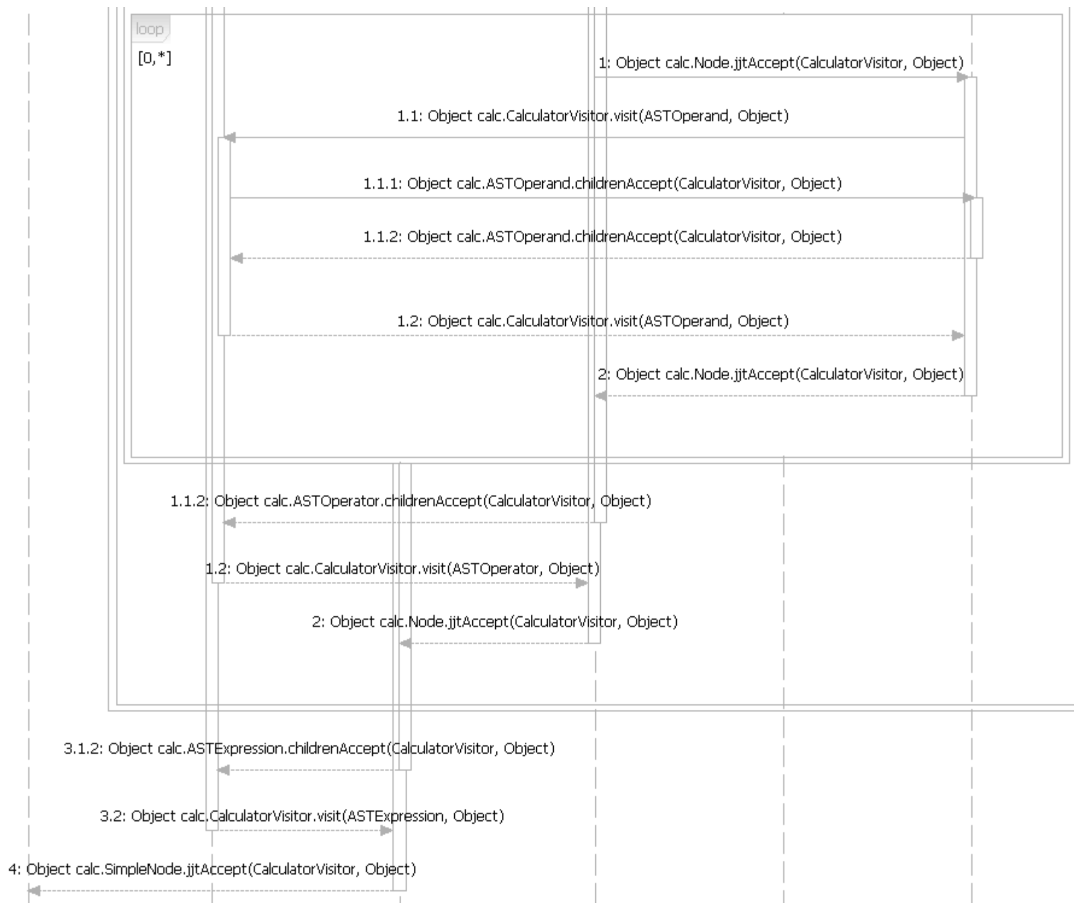

Figure 15 Reverse engineered scenario diagram for the Calculator (part II)