Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software

Lionel C. Briand, *Senior Member*, *IEEE*, Yvan Labiche, *Member*, *IEEE*, and Johanne Leduc

Abstract—This paper proposes a methodology and instrumentation infrastructure toward the reverse engineering of UML (Unified Modeling Language) sequence diagrams from dynamic analysis. One motivation is, of course, to help people understand the behavior of systems with no (complete) documentation. However, such reverse-engineered dynamic models can also be used for quality assurance purposes. They can, for example, be compared with design sequence diagrams and the conformance of the implementation to the design can thus be verified. Furthermore, discrepancies can also suggest failures in meeting the specifications. Due to size constraints, this paper focuses on the distribution aspects of the methodology we propose. We formally define our approach using metamodels and consistency rules. The instrumentation is based on Aspect-Oriented Programming in order to alleviate the effort overhead usually associated with source code instrumentation. A case study is discussed to demonstrate the applicability of the approach on a concrete example.

Index Terms—UML, sequence diagram, reverse engineering, distribution, RMI, AspectJ, OCL.

1 Introduction

To fully understand an existing object-oriented system (e.g., a legacy system), information regarding its structure and behavior is required. This is especially the case in a context where dynamic binding and polymorphism are used extensively, or when the system under study is multithreaded or distributed. 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 [28], [29], the class, sequence, and statechart diagrams can be (partially) reverse-engineered. Besides helping comprehension, our motivation is to use such diagrams to help quality assurance and during testing as test models and test oracles. For example, it is useful in practice to compare them to sequence diagrams found in the Analysis or Design documents, the objective being to find discrepancies between the two versions and thereby detect failures or design problems.

Reverse-engineering capabilities for the static structure (e.g., the class diagram) of an object-oriented system are already available in many UML CASE tools [5], [19]. However, some challenges still remain to be addressed, such as how to distinguish between plain association, aggregation and composition relationships, and the reverse engineering of to-many associations [22].

- *L.C. Briand and Y. Labiche are with the Software Quality Engineering Laboratory, Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada. E-mail: {briand, labiche}@sce.carleton.ca.*
- *J. Leduc is with Siemens Corporate Research Inc., 755 College Road East, Princeton, NJ 08540. E-mail: johanne.leduc@siemens.com.*

Manuscript received 9 Dec. 2005; revised 3 Apr. 2006; accepted 20 July 2006; published online 27 Sept. 2006.
Recommended for acceptance by T. Gyimothy and V. Rajlich.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0324-1205.

Reverse engineering and understanding the behavior of an object-oriented system is even more difficult than understanding its structure. 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. Multithreading and distribution further complicate analysis. It is then difficult to follow program execution and produce a UML sequence diagram. Similarly, identifying method call sequences from source code requires complex techniques, such as symbolic execution, in addition to source code analysis and is not likely to be applicable in the case of large and complex systems [15] due to, for example, the problem of identifying infeasible paths in interprocedural 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. Unfortunately, the accuracy of a reverse-engineered dynamic model depends on how extensively one observes runtime behavior. Though a static analysis can present a complete picture of what could happen at runtime, it does not necessarily show what actually happens. It thus appears desirable to focus on a synergy between static and dynamic analysis techniques in future research endeavors. Replacing our work in context, we focus in this article on one problem, using one kind of analysis technique: the reverse-engineering of distributed communications using a dynamic analysis technique.

Any dynamic analysis aimed at reverse engineering UML sequence diagrams (as well as any other kind of dynamic model), then, must address six separate but complementary issues. First, an *instrumentation* strategy has to be devised to gather, at runtime, the necessary information to generate sequence diagrams, while reducing to the maximum extent possible the impact on execution

and the human overhead usually associated with instrumentation. Note that the kind and amount of information to be gathered during execution—in other words, the instrumentation strategy—is driven by the subsequent steps which determine the kind of information required to obtain complete and correct sequence diagrams at the needed level of detail. A second important issue is to define a *logging* strategy to store, in an appropriate format, the data produced when executing the instrumented system. Third, execution traces must then be processed to identify the executed use case scenarios that are subsequently modeled using 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 alternatives for a use case. A fourth issue, denoted as *merging*, is to build a complete sequence diagram, for a given use case, from a set of scenario diagrams. This requires triggering as many scenarios as possible (e.g., using black-box testing techniques) through multiple executions of the system, and their analysis to merge them into one sequence diagram. Fifth, as discussed above, in the case where the reverse-engineered sequence diagrams are used as test oracles, a *comparison* procedure must exist to compare reverse-engineered sequence diagrams to design sequence diagrams, and find discrepancies (e.g., different message sequences). Last, a *visualization* technique must be devised to effectively display reverse-engineered sequence diagrams.

To investigate how to make instrumentation less intrusive, we will explore the use of Aspect-Oriented Programming (AOP) [12] to support the instrumentation of Java bytecode and discuss why AOP is a promising technology for our purpose. This paper will also cover the logging strategy and the derivation of scenario diagrams from traces. Due to size constraints, this article covers a subset of the above issues and mostly focuses on issues related to distribution, an important aspect of most modern software systems. The instrumentation of threads (active classes in UML) is left out and material regarding this important aspect can be found in [6].

One important methodological challenge comes from the fact that the sequence diagrams produced during analysis and design are not straightforward representations of the traces generated during the execution of the system. For example, the conditions under which calls are executed are reported in the scenario diagram and repetitions of message(s) are identified (if a message is executed several times, it appears only once within a loop combined fragment). Additionally, issues related to distribution have to be addressed: For instance, we need a way to uniquely identify objects executing on different nodes in the network to precisely identify message senders and receivers. To formalize our approach and specify it from a logical standpoint so that it can be analyzed and compared in future research works, we define two metamodels (using class diagrams), one for traces and another for scenario diagrams, and define mapping rules between them using the Object Constraint Language (OCL) [41]. These rules are then used as specifications to implement a tool to instrument code so as to generate traces and transform the traces into scenario diagrams.

This paper is structured as follows: An extensive and structured analysis of related works is provided in Section 2. Our approach is then detailed in Sections 3 (producing a scenario diagram from a trace) and 4 (our instrumentation strategy). We then illustrate the approach on a case study (Section 5). Conclusions and future research directions are provided in Section 6.

2 Related Work

The scientific literature is very rich with proposals and techniques aimed at reverse-engineering dynamic models and, in particular, interaction diagrams (diagrams that show objects and their interactions). Two categories of related works are relevant to our approach: Strategies targeting nondistributed systems (Section 2.1), based on some either dynamic or static analysis, and strategies targeting distributed systems (Section 2.2).

2.1 Understanding Nondistributed Systems

2.1.1 Dynamic Analysis

As for understanding nondistributed systems using dynamic analysis, differences between existing approaches are summarized in Table 1. Though not exhaustive, this table does illustrate the differences relevant to our work by means of seven criteria:

- *Granularity of the analysis, at the class or object level.* At the class level, it is not possible to distinguish the (possibly different) behaviors of different objects of the same class. In [7], [31], the memory addresses of objects are retrieved to uniquely identify them, though (symbolic) names are usually used in interaction diagrams. (This work is an extension of [7].) It seems that, in [31], methods that appear in an execution trace are not identified by their signature, but by their name (parameters are omitted), thus making it difficult to differentiate calls to overloaded methods. In the simple example they use, interacting objects can easily be identified as they correspond to attributes and as there is no aliasing. In [27], objects are identified by numbers, though nothing is said on how those numbers are determined.
- *Strategy used to retrieve dynamic information.* It can include source code instrumentation, the instrumentation of a virtual machine, or the use of a customized debugger. In the case of Rational Test Real-Time [19], the strategy used to retrieve dynamic information is proprietary information.
- *Target language.* Though this criterion should not affect the fundamental principles according to which dynamic information is retrieved, it has an impact on the instrumentation technology that can be used.
- *Control flow information and conditions.* This is the flow of control in methods and the conditions corresponding to the flows of control actually executed. Note that in [37], as mentioned by the authors, it is not possible to retrieve the conditions corresponding to the flow of control since they use a debugger: The information provided is simply the line number of control statements. Though not mentioned in the article, this limitation may also apply to [27].
- *Technique used to identify patterns of execution.* The objective is to identify sequences of method calls that repeat in an execution trace. The authors in [9], [20], [31], [37] aim to detect patterns of executions resulting from loops in the source code. However, it is not clear, due to lack of reported technical details and case studies, whether detected patterns can distinguish the execution of loops from

TABLE 1
Related Work for Nondistributed Systems, Using Dynamic Analysis

	Class vs. Object level	Information source	Language	Control flow	Conditions	Patterns	Models produced
[20]	Class	Source code instrumentation	C++	No	No	String matching (heuristics)	MSC
[40]	Class	Virtual machine	Smalltalk	No	No	No	Custom diagrams
[37]	Class	Customized debugger	Java	Yes	No	String matching	UML SD-like
[31]	Object (memory address)	Source code instrumentation	Smalltalk	No	No	Provided by user	UML SD
[9]	Object	Virtual machine	Java	No	No	Recurrences of calls	UML SD-like
[27]	Object	Java debug interface	Java	No	No	No	UML SD
[19]	Object	Proprietary information	C++, Ada, Java	No	No	N/A	UML SD
[7]	Object (memory address)	Source code instrumentation	C++	Yes	Yes	Loops	UML SD

incidental executions of identical sequences in different contexts. This is especially true when the granularity of the analysis is at the class level. For instance, it is unclear what patterns existing techniques can detect when two identical sequences of calls in a trace come from two different methods of the same class (no loop is involved). On the other hand, in [7] it is possible to identify repetition of messages due to loops since those programming language constructs are instrumented.

- *Model produced.* The three types of models reported are Message Sequence Chart (MSC), Sequence Diagram (SD), and Collaboration Diagram (CD). Custom diagrams reporting objects and their communications are used in [40], thus requiring specific tool support.

2.1.2 Static Analysis

A smaller number of static analysis techniques have been reported in literature [5], [21], [32], [39]. They all generate UML collaboration [21], [39] or sequence [5], [32], [39] diagrams (the most recent one focuses on UML 2.0 [32]), account for method control flow and conditions[1] with the exception of [21], and target either Java [5], [21], [32] or C++ [39].

The main difference between these techniques is the analysis of object references in the source code and, thus, the way objects are identified and then displayed in the generated diagrams. In the simplest case, parameters, local variables, and attribute names are used to identify objects [5]. A very similar approach is presented in [21]. A similar though less coarse-grained analysis is described in [39]. The technique represents all the runtime objects potentially created by a call to a constructor (the analysis looks for `new X(...)` statements instead of variable, parameter, and attribute names) by a single object in the generated sequence diagram. This, however, results in (possibly highly) inaccurate diagrams as one object in a generated sequence diagram can then represent many runtime objects [32].

A more sophisticated analysis of the source code to identify the different objects involved in interactions is presented in [32], [33]. In particular, the technique is able to recognize when one call site always refers to the same runtime object (referred to as *singleton call site* by the authors), or when two different call sites do so (the notion of *equivalent classes* for singleton class sites), in which case, only one diagram object is displayed, thus resulting in an accurate diagram. The analysis has, however, some limitations, as reported by the authors' experiments: For a given program, they observed that the percentage of call sites for which it is possible to uniquely identify a runtime object can range from 56 percent to 100 percent, depending on the specifics of the source code.

Last, a static analysis for the reverse engineering of sequence diagrams for Web applications is described in [10]. Not enough technical details are provided though: The approach is based on a heuristic which does not appear to be automatable.

2.1.3 A Comparison of Static and Dynamic Analyses

Static and dynamic approaches for the reverse engineering of interaction diagrams both have a number of advantages and drawbacks. Most notably, a dynamic approach can report precisely on objects actually interacting, whereas the accuracy of a static analysis (even the currently most powerful ones [32]) may vary a great deal. On the other hand, the accuracy of a dynamic approach highly depends on the inputs being used to execute the system, whereas a static approach reports on a complete picture of what could happen at runtime, though this may include infeasible behavior. Though black-box and white-box test techniques [3] can help increase the accuracy of a dynamic approach, we can never be sure of the completeness of the dynamic analysis. A dynamic approach also relies on the merging of several traces, each reporting on one observed behavior, and abstraction mechanisms (e.g., abstracting remote communications as reported in this paper, abstracting

1. Although this is not explicitly described in [39], the paper suggests that the tool implementing the approach reports on the control flow and conditions.

TABLE 2
Related Work for Distributed Systems

	Component vs. Object level	Information source	Language	Thread information	Time issue	Model produced
[2]	Component	Data stream (instrumentation)	Java	No	Trace history + timestamp	UML SD
[23]	Object	Source code instrumentation	C/C++, Java	No	RFC2030	Trace, temporal constraints
[25]	Component	Remote procedure call (IDL)	CORBA	NA	Time compensation	Performance statistics
[38]	Component	Remote procedure call (IDL)	CORBA	NA	Trace history + timestamp	Trace
[34]	Object	JVM profiler	Java	Yes	Logical time	UML SD

asynchronous communications as reported in [6]) to cope with the large amount of data.

This clearly calls for more research studying the synergy between static and dynamic approaches, similar to what has recently been suggested in [17]. In particular, we believe that a static analysis would be very helpful when different scenario diagrams produced by a dynamic analysis have to be merged to generate a sequence diagram (the merging step we mentioned in the Introduction).

2.2 Understanding Distributed systems

To the best of our knowledge, a smaller number of approaches address the reverse engineering of dynamic information for distributed or real-time systems. The five approaches discussed in this section are compared according to six criteria (see Table 2). Note that none of these approaches provide information on the control flow (or conditions) or recognize repetitions of message sequences, as these aspects are not their main focus. Rational Test Real-time [19] requires execution on one computer and thus does not allow the tracing of distributed behavior. The six criteria are

- *Granularity of the analysis, at the component or object level.* We use the term component here, rather than class, since approaches for distributed systems tend to focus on remote calls between components rather than interclass communication, as in the previous section. They consider components executing on nodes in a distributed environment and those components usually correspond to executables of logical subsystems plus associated files and data. This difference is in part due to the source of information used: The strategies solely based on distribution middleware (e.g., streams in Java RMI,[2] interceptors in CORBA) are inherently confined to providing information on components [2], [25], [38].
- *The strategy used to retrieve dynamic information.* It can include source code instrumentation [23], a JVM profiler [34], data stream communications between distributed objects [2], or CORBA interceptors that provide a hook at the remote procedure call level [25], [38]. Note that, though data stream communications between distributed objects are traced in [2], the authors mention that they do not distinguish different instances of the same class, thus our classification as "component." Additionally, the authors suggest providing specific implementations to Java classes `OutputStream` and `InputStream` as these classes are used for network communication using RMI, thus requiring source code instrumentation to make sure those specific implementations are actually used. It is also worth mentioning that the information extracted from CORBA interceptors may vary with the Object Request Broker (ORB) implementation. Last, in [23], the authors define a library of C/C++ functions called `rlog` to log data in a distributed environment, thus also requiring manual source code instrumentation. Since this approach requires that the user knows exactly what to instrument, it seems that `rlog` can be used to retrieve information on the control flow, for instance, though this is not mentioned by the authors.
- *Target language.* Approaches based on the CORBA middleware are only based on the Interface Definition Language and can thus be used for distributed components implemented in a variety of languages, such as C, C++, and Java. Note that since `rlog` is only a library of functions, it can be used in a Java program.
- *Threads and timing issues.* Generating a dynamic model showing distributed objects interactions, such as a UML sequence diagram, requires that messages be ordered, within or between threads executing on a computer, but also between threads executing on different computers. However, in a distributed system, there is often no global clock that could be used to order messages gathered from different computers. In [23], time offsets between computers are calculated based on RFC 2030.[3] The authors of [25], [34] use techniques that have been proposed in the literature to capture causality between events of a distributed system [18], [30]. The other two approaches do not provide enough information with respect to the time issue: In [2], [38], the authors use trace histories and timestamp, and mention causal relationships between events, respectively, but the descriptions lack details.

2. Java Remote Method Invocation (RMI) is the Java communication middleware that basically provides Remote Procedure Calls mechanisms in Java.

3. This document describes the Simple Network Time Protocol (SNTP) Version 4, which is used to synchronize computer clocks in the Internet [24].

- *Model produced.* Only two approaches provide UML sequence diagrams [2], [34]. In [34], a sequence diagram corresponds to a thread, though the implementation of a sequence diagram, as defined during Analysis or Design can involve several threads. The others only generate trace data and provide mechanisms to produce performance statistics [25] or check temporal constraints [23]. None of them attempt to abstract out details due to the communication middleware.

2.3 Conclusion

The discussion above suggests that a complete strategy for the reverse engineering of interaction diagrams (e.g., a UML sequence diagram) in a distributed context should provide information on 1) the objects (and not only the classes or components) that interact, provided that it is possible to uniquely identify them; 2) the messages these objects exchange, which are characterized by their corresponding invocations being identified by method names and actual parameters' values and types. Note that messages can be synchronous or asynchronous and that communicating objects can be located on different nodes of the network. In the asynchronous case, messages are characterized by a signal [4], [13] and labeled with the signal name (i.e., threads communicate through signals); 3) the control flow involved in the interactions (branches, loops, exceptions), as well as the corresponding conditions. Last, such a strategy should attempt to abstract out details due to the communication middleware.

None of the approaches we have discussed here covers all the above information pieces and one of the goals of the research reported in this paper is to address these issues in a way which is the least intrusive possible for developers and testers. Another issue we tackle in this paper, which is more methodological in nature, is how to precisely express the mapping between traces and the target model. Many of the papers published to date do not precisely report on such mapping so that it can be easily verified and built upon. Partial exceptions are [7], [21] in a nondistributed context and [34] in a distributed context, where metamodels are defined for traces. In [34], a simple trace metamodel is provided in an other formalism than UML, and no mapping is described. In [21], a UML trace metamodel is proposed and a mapping is informally described. Our strategy in this paper has been to define this mapping in a formal and verifiable form as consistency rules between a metamodel of traces and a metamodel of scenario diagrams,[4] so as to ensure the completeness of our metamodels and enable their verification.

Note that one important question for the reverse engineering of UML 2.0 sequence diagrams is, as mentioned in [33], whether UML 2.0 is a powerful enough notation to represent all possible control flow that can be reverse engineered from source code. As described in [33], this is not the case, especially in the presence of certain unstructured programming constructs, and the authors have suggested extensions to the UML 2.0 notation to represent specific source code construct. For example, a break statement can break out of more than one loop in the code, whereas the UML 2.0 break combined fragment only breaks out of the enclosing loop.

4. Consistent with the UML standard [28], the term metamodel is used here to denote a class diagram whose instance represents a trace or scenario diagram, i.e., a model of the system behavior.

3 From Runtime Information to Scenario Diagrams

Our high-level strategy for the reverse engineering of scenario diagrams in a distributed context consists in instrumenting the System Under Study (SUS), executing the instrumented SUS (thus producing traces), and analyzing the traces in order to reverse-engineer scenario diagrams while addressing the issues mentioned in the previous section. In this paper, we assume the SUS is implemented in Java and uses RMI as distribution middleware. However, the conclusion will discuss why many components of the approach can be adapted to other programming languages and middleware platforms. Furthermore, because our target notation is UML 2.0, we can only reverse-engineer programming constructs that can be represented in the current definition of the standard. As discussed in the previous section, some unstructured programming constructs (e.g., goto-like statements breaking out of multiple loop levels) cannot be represented and our working assumption is, therefore, that they are not present in the code. Extending the UML 2.0 standard to account for them is an entire research project in itself.

We first devise a metamodel of scenario diagrams that is an adaptation and simplification of the UML 2.0 metamodel for sequence diagrams (Section 3.1). This helps us define the requirements in terms of information we need to retrieve from the traces, which will then drive our instrumentation (Section 4). We formalize these requirements as a metamodel of traces (Section 3.2). These metamodels are then used as follows: The execution of the instrumented SUS produces a trace, which is transformed by our tool into an instance of the trace metamodel. This trace metamodel instance is then transformed into an instance of the scenario diagram metamodel, using algorithms which are directly derived from consistency rules (or constraints) we define between the two metamodels (Section 3.3). Those rules are described in the Object Constraint Language (OCL) [41] and are useful in the following ways: 1) They provide a logical specification and guidance for our transformation algorithms that derive a scenario diagram from a trace (both being instances of their respective metamodels), 2) they help us ensure that our metamodels are correct and complete, as the OCL expression composing the rules must be consistent with the metamodels. OCL is a natural choice as it offers numerous constructs (e.g., collection operations) to simplify the definition of constraints on UML class diagrams.

3.1 Scenario Diagram Metamodel

Sequence diagrams [29] are among the crucial diagrams used during the analysis and design of object-oriented systems, as they are used to identify object responsibilities and interactions associated with each use case [8]. A sequence diagram describes how objects interact with each other through message sending, and how those messages are sent, possibly under certain conditions, in sequence. We have adapted the UML 2.0 metamodel [28], that is, the class diagram that describes the structure of sequence diagrams, to our needs, so as to simplify the specification of consistency rules and the generation of scenario diagrams from traces.

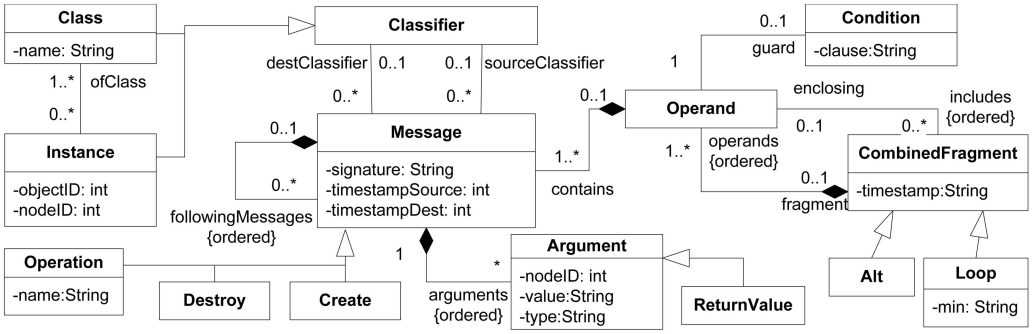

Fig. 1. Scenario diagram metamodel.

Our scenario diagram metamodel is shown in Fig. 1. Messages (abstract class `Message`) have a source and a target (role names `sourceClassifier` and `destClassifier`, respectively), both of type `Classifier`. The source and destination objects of a message can be named objects (class `Instance`), or classes (class `Class`) in the cases where class scope methods are executed. A message can be an `Operation` call or can correspond to the creation or destruction of an object (classes `Create` and `Destroy`). Messages can have arguments (class `Argument`) of different types (attribute `type`), i.e., primitive types or object types. Depending on the type, the other attributes of `Argument` provide additional information: In the case of a primitive type, attributes `value` and `type` are self-explanatory. In the case of an instance of a user-defined class, `type`, `value` and `nodeID` are used to uniquely identify the instance in the instrumented distributed system, `value` captures the unique identification (for a given class and node) of the instance and `nodeID` uniquely represent nodes in the network, as further described in Section 4.

Messages can be triggered under certain conditions. In a UML 2.0 sequence diagram, such messages are part of a *combined fragment* (compositions between `CombinedFragment` and `Operand`, and between `Operand` and `Message`) [28]. In our context, a combined fragment can have one of the following *operators*: `alt`, `loop`. It is composed of one or more `operand`s (subfragments). Each operand is characterized by zero or one condition (association to `Condition`). Furthermore, combined fragments can be nested (association between `Operand` and `CombinedFragment`) into the operand of another combined fragment, e.g., nested `if` or `switch` statements. Iterations of messages are modeled by `loop(min, max)` fragments in UML and are modeled as the subclass `Loop` in our metamodel with a corresponding `min`. The `max` attribute cannot be determined through the reverse engineering of a single scenario and it is therefore not included in our metamodel. A `Loop` fragment repeats as long as the guard condition is true when checked at the top of the loop body (association `guard`). Similar to `Loop`, `Alt` is a subclass of `CombinedFragment` and is associated to one or more operands and guard conditions. Other types of UML 2.0 combined fragments are not included here for diverse reasons. The `opt` operator is semantically equivalent to an `alt` with only one `operand`. The UML 2.0 `break` would model interactions triggered by `break` statements or exception handling in our context. However, such interactions are usually not part of design sequence diagrams and when they are, they do not get included until late design stages (object design in [8]). We therefore do not address exceptions in this work and `break` statements are further discussed in Section 4.4. Other types of combined fragments are not considered as either they are not needed when modeling single scenarios because we do not address threads in this paper (e.g., `par`), or because their purpose is to add information to sequence diagrams at higher levels of abstraction (e.g., `critical`, `negative`, `assert`).

Last, a message can trigger other messages: `{ordered}` self association on class `Message`. The order of messages and their possible grouping in repetitions is devised using timestamps (in class `Message`) that allow us to order messages exchanged between objects in the distributed SUS. `timestampSource` and `timestampDest` in class `Message` refer to the sending and receiving of the message, respectively. Timestamps are further discussed in Section 4.1.

3.2 Trace Metamodel

We instrument the SUS by processing the source code and the bytecode and adding specific statements to the bytecode to retrieve the required information at runtime (Section 4). These statements are automatically added and produce text lines (referred to as trace statements) in the trace file, reporting on the following:

1. Method entry and exit. The method signature, the class of the target object (i.e., the object executing the method), unique identifiers (in the distributed SUS) for the target object, and the arguments are reported.
2. Conditions. For each condition statement, the kind of statement (e.g., "if," "case," ...,) and the condition as it appears in the source code are reported;
3. Loops. For each loop statement, the kind of the loop (e.g., "while"), the corresponding condition as it appears in the source code, and the end of the loop are reported;
4. Distribution information. RMI remote calls (in client) and executions (in server) are reported.

Note that in each case, a `timestamp` (based on each node's local time) indicating when the event occurred is also reported in the trace (Section 4.1 discusses why no global timestamps are necessary).

From the trace files, it is possible to instantiate the class diagram in Fig. 2, which is the metamodel for our traces. This class diagram has similarities to our scenario diagram metamodel but there are some important differences: For instance, a `Message` object has direct access to its source and target objects (instances of `Classifier`) in the scenario diagram metamodel (Fig. 1), whereas a `MethodExecution` has only access

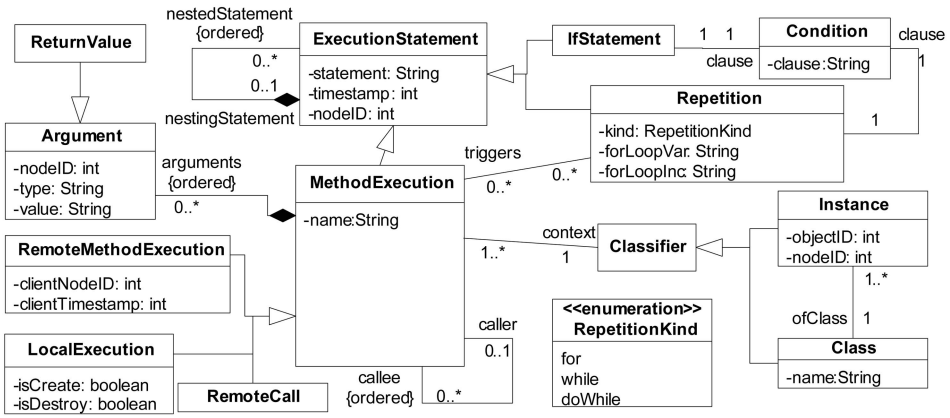

Fig. 2. Trace metamodel.

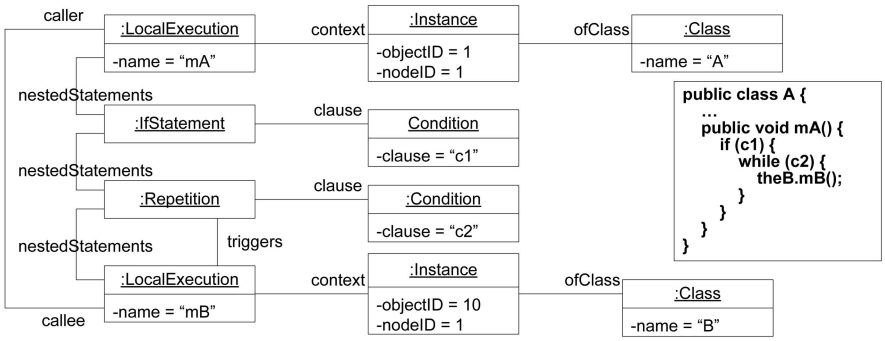

Fig. 3. Instance example of the trace metamodel (excerpt).

to the object that executes it, called the `context` (i.e., the target of the corresponding message) and has to query the method that called it (self association on `MethodExecution` with role name `caller`) to identify the source of the corresponding message (Fig. 2). This `caller-callee` information, though not directly available in the trace file (i.e., when reporting a method execution as a trace statement, the caller is not provided), can be determined offline by analyzing the trace using the self association on `ExecutionStatement`. This association captures, for a given trace statement, the related statement it is nested in (e.g., a method invocation statement directly nested in an "if" statement) or the statements that are directly nested into it (see the example in Fig. 3). Typically, given an instance of `MethodExecution`, say, `me`, any instance of `MethodExecution` in collection `me.nestedStatement` is in collection `me.callee`. Additionally, if instances of `Repetition` or `IfStatement` are in `me.nestedStatement` and have `MethodExecution` instances in their `nestedStatement` collection, these `MethodExecution` instances are also in `me.callee`. In other words, determining the contents of `me.callee` amounts to recursively (transitive closure) detecting `MethodExecution` instances when navigating `nestedStatement` starting from `me`. A similar association exists between classes `Repetition` and `MethodExecution`, with role name `triggers`: It represents all the calls (i.e., `MethodExecution` instances) that are triggered within a given `Repetition` instance. Again, this information is redundant since it can be retrieved with a recursive traversal of association `nestedStatement`. However, it has been added to the metamodel since, as we will see in Section 3.3, it will simplify the definition of consistency rules.

Consider for example the code fragment for method `mA()`, in class A of Fig. 3. This figure also shows an excerpt of the trace metamodel instance (object diagram) of an execution of `mA()` on an instance of class A, assuming condition `c1` is true and the while loop is executed only once. An instance of `LocalExecution` is created for the execution of `mA()`: It has a context (i.e., an instance of class `Instance`) which is of type `A`. This `LocalExecution` instance has only one `nestedStatement`, i.e., an instance of class `IfStatement` (which clause is `c1`). The `IfStatement` instance has one `nestedStatement`, i.e., an instance of class `Repetition` (which clause is `c2`). An instance of `LocalExecution` is the only `nestedStatement` of the `Repetition` instance: the execution of method `mB()` on an instance of class B. The `nestedStatements` links imply a `caller-callee` link between the two `LocalExecution` instances (method executions where the caller and callee are on the same node).

The trace metamodel includes information on local method calls, namely, instances of `LocalExecution`, creation or destruction of objects. It also includes information on RMI remote calls: Class `RemoteCall` for the identification of the call on the client side and class `RemoteMethodExecution` for the identification of the actual execution on the server side. These classes' `context` (inherited from `MethodExecution`) is defined as their `caller` and `callee`'s `context`, respectively. The main reason is that, in the context of RMI, their actual context would be a stub (client side) and a skeleton (server side) in the bytecode. We are, however, interested in the objects of the SUS source code and not those introduced by the middleware during compilation and execution.

```
1 T::MethodExecution.allInstances->forAll(  me1: T::MethodExecution,
2                                           me2: T::MethodExecution |
3   CheckMapping.mapping(me1,me2)->notEmpty() implies
4   S::Message.allInstances()->exists(mes: S::Message |
5     //comparing timestamps
6     mes.timestampSource = me1.timestamp
7     and mes.timestampDest = me2.timestamp
8     //the context of me1 is the source of message mes
9     and CheckMapping.mapContextClassifier(me1.context, mes.sourceClassifier)
10    //the context of me2 is the target of message mes
11    and CheckMapping.mapContextClassifier(me2.context, mes.destClassifier)
12    //compare arguments (matching entire sequences)
13    and CheckMapping.mapExecutionMessageArgs(me2, mes)
14    //mapping the message guard
15    and
16      if (mapping(me1,me2).nestingStatement.oclIsTypeOf(MethodExecution)) then
17        mes.operand->isEmpty()
18      else // the nesting statement type is either IfStatement or Repetition
19        mes.operand.guard.clause =
20             mapping(me1,me2).nestingStatement.clause.clause
21      endif
22    //mapping the exact message type, along with message name and signature
23    and mes.signature = me2.statement
24    and mes.name = me2.name
25    and me2.isCreate = true implies mes.oclIsTypeOf(S::Create)
26    and me2.isDestroy = true implies mes.oclIsTypeOf(S::Destroy)
27    and not (me2.isCreate = true or me2.isDestroy = true) implies
28             mes.oclIsTypeOf(S::Operation)
29  ) // S::Message.allInstances->exists
30)
```

Fig. 4. Mapping `T::MethodExecution` instances to `S::Message` instances.

As discussed above, the mapping between the two metamodels is not straightforward as the information required to create instances of the scenario diagram metamodel are often not readily available in trace statements (more than one statement is required) and the instance of the scenario diagram to be generated is an abstraction of the trace metamodel instance (e.g., the scenario metamodel does not include information on the distribution middleware). It is therefore necessary to introduce precise mapping rules. We do so by defining consistency rules between the Scenario and Trace metamodels using the OCL.

3.3 Consistency Rules

We have derived five consistency rules, expressed in the OCL, that relate an instance of the trace metamodel to an instance of the scenario diagram metamodel. Note that these OCL rules only express constraints between the two metamodels. They are not algorithms, but they provide a specification and insights into how to implement such algorithms. In other words, those OCL expressions can be considered the postcondition of a single operation responsible for transforming an instance of the trace metamodel into an instance of the scenario metamodel. We provide here simple, fictitious examples to illustrate the rules, and more complicated examples of scenario metamodel instances obtained from trace metamodel instances can be found in [6]. Three consistency rules have been defined to match `Message` child classes from instances of `MethodExecution` child classes (Section 3.3.1), one consistency rule has been defined to identify links between `Message` instances, i.e., association `followingMessage` (Section 3.3.2), and one consistency rule has been defined for repetitions of `Message` instances (Section 3.3.3). In the discussion below, to denote the metamodel to which a class belongs, we use the prefixes S and T for the Scenario and Trace metamodels, respectively.

3.3.1 From `MethodExecution` Child Classes to `Message` Child Classes

The first consistency rule describes the mapping between instances of `Trace::MethodExecution` child classes and instances of `Scenario::Message` child classes (Fig. 4). (OCL keywords appear in bold face.) The most common situation occurs when two instances of `LocalExecution` are related by a `caller-callee` link (Fig. 2), as this corresponds to an instance of `S::Message` child class `Operation`. The first four lines of the consistency rule in Fig. 4 state that whenever two instances of `T::MethodExecution`, `me1` and `me2`, satisfy a given condition,[5] there exists a corresponding instance of `S::Message`. The condition is modeled as a query operation to simplify and improve the readability of our OCL expressions, namely, `mapping(me1, me2)`. This operation has two parameters of type `MethodExecution`, and its postcondition can be found in Fig. 6. In our tool prototype, it is implemented in utility class `CheckMapping`, which does not appear in our metamodel, but which will be referred to in OCL expressions.

Operation `mapping(me1, me2)` returns `me2` when `me1` and `me2` are instances of `LocalExecution` and there is a `caller-callee` link between them (i.e., `me1` calls `me2`), which clearly results in a `Message` instance (lines 6–7 in Fig. 6). Such an example was already illustrated in Fig. 3 with methods `mA` and `mB`. When `me1` and `me2` are instances of `LocalExecution` and `me1` performs a remote call to `me2`, then `mapping(me1,me2)` returns the instance of `RemoteCall` corresponding to this remote call, i.e., the caller of `me2` must be an instance of `RemoteMethodExecution` and the attributes of this caller must match the `RemoteCall` instance returned by `mapping(me1,me2)`

5. There are more conditions when dealing with threads, as reported in [6].

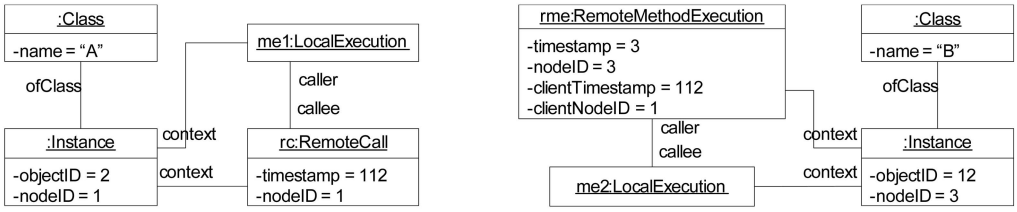

Fig. 5. Example of mapping for a remote call/execution.

```
1  context CheckMapping::mapping( le1: T::MethodExecution,
2                                 le2: T::MethodExecution ):T::MethodExecution
3  post:
4    le1.oclIsTypeOf(T::LocalExecution) and le2.oclIsTypeOf(T::LocalExecution)
5    and (
6       if ( le1.callee->includes(le2) ) then
7          result = le2      // le1 performs a local call to le2
8       else
9          result = le1.callee->select(rc:T::MethodExecution |
10                   rc.oclIsTypeOf(T::RemoteCall)
11                   and le2.caller.oclIsTypeOf(T::RemoteMethodExecution)
12                   and rc.nodeID = le2.caller.clientNodeID
13                   and rc.timestamp = le2.caller.clientTimestamp
14                )
15      endif
16   )
```

Fig. 6. Postcondition for operation `mapping()` in class `CheckMapping`.

```
1  context CheckMapping::mapContextClassifier( co : T::Classifier,
2                                               cl : S::Classifier): Boolean
3  post: result =
4    if (co.oclIsTypeOf(T::Instance) ) then (
5      cl.oclIsTypeOf(S::Instance) and cl.objectID = co.objectID
6      and cl.ofClass.name = co.ofClass.name
7      and cl.nodeID = co.nodeID
8    ) else (
9      cl.oclIsTypeOf(S::Class) and cl.name = co.name
10   ) endif
```

```
1  context CheckMapping::mapExecutionMessageArgs( me : Trace::MethodExecution,
2                                                 m : Scenario::Message): Boolean
3  post: result =
4    m.arguments.nodeID = me.arguments.nodeID
5    and m.arguments.value = me.arguments.value
6    and m.arguments.type = me.arguments.type
7    and Sequence{1..me.arguments->size()}->forAll(index: Integer |
8      if me.arguments->at(index).oclIsTypeOf(T::ReturnValue)
9        then m.arguments->at(index).oclIsTypeOf(S::ReturnValue)
10       else m.arguments->at(index).oclIsTypeOf(S::Argument) endif
11     )
```

Fig. 7. Postcondition of operation `CheckMapping::mapContextClassifier()` and `CheckMapping::mapExecutionMessageArgs()`.

(attributes `clientNodeId` and `clientTimestamp` of the `RemoteMethodExecution` instance must match the `nodeID` and `timestamp` attributes of the `RemoteCall` instance, lines 12-13). This situation is illustrated by the example trace metamodel instance in Fig. 5. Note that the instance of the trace metamodel created for the client (one trace file), on the left, is not linked in any way to the instance of the trace metamodel created for the server (another trace file), on the right. The purpose of the transformation is to abstract this situation by transforming instances `me1`, `rc`, `rme`, and `me2` into a `S::Message` instance. If no mapping is possible, i.e., there is no local call or remote call to `me2` by `me1`, then the result of `mapping(me1, me2)` is empty.

The rest of the consistency rule in Fig. 4 ensures that the attributes of `me1` and `me2`, as identified by the query operation, and the matching `Message` instance `mes` are consistent. First, `timestamps` of `me1`, `me2` and `mes` are checked (lines 6-7). Next, the contexts of `me1` and `me2` correspond to the source and target classifiers of mes, respectively. (This is checked using operation `mapContextClassifier()`, the postcondition of which is provided in Fig. 7.) The rule also ensures that the arguments of `mes` match those of `me2` (using operation `mapExecutionMessageArgs()` in Fig. 7). The mapping between `me1` or `me2` execution conditions (due to potential nesting statements) and the `mes` guard condition is then verified (lines 16-21). In the case of a local call (i.e., caller-callee relation between `me1` and `me2`), the condition triggering the execution of `me2` is given by `me2.nestingStatement.clause.clause`, assuming that `me2.nestingStatement` is a `Repetition` or an `IfStatement` (i.e., not a

```
1  Scenario::Message.allInstances->forAll( m1: Message, m2: Message |
2  (  m1.destClassifier = m2.sourceClassifier
3     and  m1.timestampDest < m2.timestampSource
4     and  Scenario::Message.allInstance->select( m: Message |
5            m.destClassifier = m1.destClassifier
6            and  m.timestampDest > m1.timestampDest
7            and  m.timestampDest < m2.timestampSource  )->isEmpty()
8  ) implies m1.followingMessages->includes(m2)
9  and
10 Scenario::Message.allInstances->forAll(m: Message |
11   Sequence{1..m.followingMessages->size()}->forAll(i: Integer, j: Integer |
12     i > j  implies  m.followingMessages->at(i).timestampSource
13                     >  m.followingMessages->at(j).timestampSource
14 )  )
```

Fig. 8. Identifying `followingMessage` links between `Message` instances.

`MethodExecution`), regardless of whether the condition is due to a repetition or an if statement. In the case of a remote call, the condition triggering the execution of `me2` is the condition triggering the execution of the remote call (instance `rc` in Fig. 5): i.e., `rc.nestingStatement.clause.clause`, assuming `rc.nestingStatement` is not a `MethodExecution` instance. The OCL expression in Fig. 4 is simplified by the fact that `mapping(...)` returns either `me2` (local call) or `rc` (remote call) or is empty and the fact that role names from `IfStatement` and `Repetition` to `Condition` are the same (`clause`). Last, the signature, name and actual type (child class of `Message`) of the message are checked (lines 23–28).

One situation is worth mentioning here: method calls that appear in conditions (e.g., in an if condition). Due to our instrumentation strategy, such a situation results, in the trace, in a `MethodExecution` instance for each method executed during the evaluation of the condition (Sections 4.2) followed by an `IfStatement` instance reporting on the complete condition expression (Section 4.4). If short-circuit evaluation is used to optimize condition evaluations, not all the method calls in conditions may result in `MethodExecution` instances in the trace: Only those method calls that appear in terms that are actually evaluated do. The scenario diagram will thus show only those method executions that actually took place during execution. It is only when several scenario diagrams are available that a complete picture will be generated, by merging them.

Two other consistency rules are necessary in this section to map `Message` instances to `MethodExecution` instances that cannot be paired with any other `MethodExecution` instance to satisfy the query operation we used (see details in [6]). Indeed, there may exist `LocalExecution` instances in the trace metamodel instance without any caller. This is the case of method `main()`: It is not called by any other method. Other typical examples are calls that originate from noninstrumented subsystems (e.g., a GUI subsystem). These `LocalExecution` instances are nevertheless mapped to `Scenario::Message` instances, though the message does not have a source classifier.

3.3.2 Identifying `followingMessage` Links

The identification of the following messages of a given message (association `followingMessages` in Fig. 1) is the purpose of a separate rule (Fig. 8). It requires that all the messages be identified, using the rules described in the previous section. Recall that self association `followingMessages` on `S::Message` specifies the ordered sequence of messages that are triggered by a given message.

The rule in Fig. 8 is a conjunction. The first conjunct (lines 2-8) identifies, for a given `Message` instance `m1`, the set of `Message` instances that are triggered by `m1` among the set of all the `Message` instances which have `m1` destination classifier as source classifier. This is performed using timestamps of `Message` instances (`timestampDest` and `timestampSource`). `Message` instance `m2` is triggered by `m1` if and only if `m1` destination classifier is `m2` source classifier (line 2), `m2` is sent after `m1` is received (line 3), and there is no other `Message` instance received by that classifier between those two timestamps (lines 4-7). The second conjunct (lines 10-14) checks that the elements of sequence `m.followingMessages`, for any `Message` instance `m`, are sorted according to their timestamps `timestampSource`.

3.3.3 Identifying Repetitions of `Message` Instances

The last rule matches instances of `T::Repetition` and instances of `S::Loop` (Fig. 9). The rule first (lines 1-2) states that any instance of `S::Loop` corresponds to an instance of `T::Repetition` that is linked to `MethodExecution` instances. Indeed, a `S::Loop` instance is linked to `S::Message` instances, and the `T::Repetition` instance must thus involve the `T::MethodExecution` instances that correspond to these `Messages`. The rest of the rule describes how the `T::Repetition` and `S::Loop` instances relate to each other, that is, how their attributes and links relate to each other: their types, timestamps, and conditions must match (lines 3-7); nested and nesting statements must match relationships between operands and combined fragments (rest of rule).

The `Message` instances associated with the `S::Loop` instance must match the `T::MethodExecution` instances nested in the `T::Repetition` instance. This is checked using query operation `getMessage(me : T::MethodExecution)`, the postcondition of which can be found in Fig. 10. The following two different cases have to be considered. First (lines 6-11), `MethodExecution` instance `me`, in the `T::Repetition`, is a `LocalExecution`. In this case, `getMessage(me)` returns the (unique) `Message` instance that corresponds to `me`. It uses `me`'s context and `timestamp` to check the message `destClassifier` and `timestampDest` and `me` caller's `context` and `timestamp` to check the message `sourceClassifier`. Second (lines 14-18), `MethodExecution` instance `me` is a `RemoteCall`.[6] In this case, `getMessage(me)` returns the (unique)

6. Note that `RemoteMethodExecution` instances cannot be triggered by `Repetition` instances since they are artificially introduced by our instrumentation procedure (i.e., wrappers), i.e., they do not correspond to executions of methods in the SUS.

```
T::Repetition.allInstances->forAll(r | r.triggers->notEmpty() implies
S::Loop.allInstances()->exists(l |
  r.kind = RepetitionKind::while implies l.min = '0'
  and r.kind = RepetitionKind::dowhile implies l.min = '1'
  and r.kind = RepetitionKind::for implies l.min = r.forLoopVar
  and r.timestamp = l.timestamp
  and r.clause.clause = l.operand->first.guard.clause
  //compare nesting dependencies
  and if ( r.nestingStatement.oclIsTypeOf(IfStatement) or
           r.nestingStatement.oclIsTypeOf(Repetition)     ) then
    l.enclosing.fragment->notEmpty()
    and r.nestingStatement.oclIsTypeOf(IfStatement)
          implies  l.enclosing.fragment.oclIsTypeOf(Alt)
    and r.nestingStatement.oclIsTypeOf(Repetition)
          implies  l.enclosing.fragment.oclIsTypeOf(Loop)
    and r.nestingStatement.clause.clause = l.enclosing.guard.clause
    and l.enclosing.includes->includes(l)
    and l.enclosing.fragment.operand->includes(l.enclosing)
  endif
  and r.nestedStatement->forAll(ns | ns.oclIsTypeOf(IfStatement)
        implies  l.operand.contains->exist(cf | cf.oclIsTypeOf(Alt)
                   and  cf.operand.condition.clause = ns.condition.clause) )
  //compare Messages/MethodExecutions in Loops/Repetitions
  and r.nestedStatements->select(MethodExecution)->forAll(me |
    l.operand.message->includes(CheckMapping.getMessage(me))
    and CheckMapping.getMessage(me).operand.fragment = l )
  ) // end of exists(...)
) // end of forAll(...)
```

Fig. 9. Mapping `T::Repetition` to `S::Loop`.

```
CheckMapping::getMessage(me: T::MethodExecution): S::Message
post:
let rme:RemoteMethodExecution = RemoteMethodExecution.allInstances->select(r|
           r.clientNodeID = me.nodeID and r.clientTimestamp = me.timestamp)
in
  me.oclIsTypeOf(LocalExecution) implies
    result = Message.allInstances->select(m:Message |
      mapContextClassifier(me.context, m.destClassifier)
      and  mapContextClassifier(me.caller.context, m.sourceClassifier)
      and  m.timestampDest = me.timestamp
      and  m.timestampSource  = me.caller.timestamp  )->asSequence->at(1)
  and
  me.oclIsTypeOf(RemoteCall) implies
    result = Message.allInstances->select(m:Message|
      mapContextClassifier(rme.callee.context, m.destClassifier)
      and  mapContextClassifier(me.context, m.sourceClassifier)
      and  m.timestampDest = rme.callee.timestamp
      and  m.timestampSource = me.caller.timestamp  )->asSequence->at(1)
```

Fig. 10. Postcondition of operation `CheckMapping::getMessage()`.

`Message` instance that corresponds to `me`, using `me`'s `context` and `timestamp` and the `context` and `timestamp` of the `RemoteMethodExecution` corresponding to `me`.

It is worth mentioning that, following our instrumentation strategy for control flow structures (next section), multiple iterations of a loop in the code result in the same number of instances of `Repetition` in the `Trace` metamodel instance. Since the rule in Fig. 9 establishes a one to one mapping between `T:Repetition` and `S:Loop` instances, multiple iterations of a loop in the code result in as many instances of `Loop` in the scenario metamodel instance (assuming the `Repetition` instances trigger method calls, as specified in Fig. 9) and thus in as many combined fragments in the displayed scenario diagram. These `Loop` instances, however, occur one after the other in the scenario (i.e., no intermediate `Message` between them) and all have the same condition (the one retrieved from the code). It is thus not difficult to automatically recognize, from a scenario, that this sequence of combined fragments is the result of multiple iterations of a single loop in the code. Combining these related combined fragments into one single `Loop` combined fragment entails merging their triggered message sequences, which is part of the merging step not covered in this paper.

4 Instrumentation

Instrumenting the source code poses a number of problems. The user faces a dilemma: Keep only the clean (original) version or the instrumented version or keep both versions of the source code. Both options have disadvantages. To avoid repeated instrumentation and cleaning of the source code and possible inconsistencies between the instrumented and noninstrumented versions, we aim at using a less intrusive instrumentation strategy, and thus we use AOP

[12] to support the instrumentation of Java systems' bytecode and, more specifically, AspectJ [16].

Another limitation of instrumentation is the effects that the added code might have on the execution of the SUS: because of delays introduced by the execution of the instrumentation code, the behavior of the SUS may be different from the expected one (e.g., deadlines). This is an issue that is unavoidable; observing a system changes the system [35]. However, we aim at minimizing execution overhead by instrumenting as few constructs as possible while still obtaining the needed information (recall our metamodels described in Sections 3.1 and 3.2). Unless the SUS is a hard real-time system with deadlines, the delays introduced by the instrumentation should not change the intended behavior of the system, if proper synchronization techniques are used.

As mentioned in previous sections, the final objective of our approach is to reverse-engineer UML sequence diagrams, and the instrumentation strategy, i.e., what has to be observed at runtime, is thus driven by the information required to draw those diagrams (recall the Scenario and Trace metamodels). More specifically, we are interested in sequences of synchronous messages (including arguments and return values) exchanged between objects and the conditions or repetitions driving them. Additionally, we would like the sequence (and, thus, scenario) diagrams to be abstractions of the implementation and hide low-level design details, for instance, related to the middleware used for distribution. (Note that parts of our approach to reverse engineer asynchronous messages exchanged by threads are not reported in this article due to space constraints, but can be found in [6].) The objective is to provide sufficiently high-level diagrams that would help the user better understand class responsibilities and communication within use case executions without having to cope with unnecessary details. The context in which we operate assumes synchronous Remote Method Invocation (RMI) mechanisms for communication between separate JVMs (that are possibly on separate machines), referred to as nodes in this work. Recall that, since we reverse-engineer UML 2.0 sequence diagrams, we are only interested in what can be represented with such diagrams. As described in [33], doing so leads to potentially inaccurate sequence diagrams because of differences between the UML 2.0 notation and Java constructs. In particular, a Java `break` statement can break out of multiple nested loops, whereas a UML 2.0 break combined fragment only breaks out of the immediately enclosing loop. A Java `break` statement can even branch to any location in a method body thanks to labels (this is then similar to a `goto`). The authors of [33] suggested extensions to the UML 2.0 notation to address these issues and reported, by means of 20 different case study programs, that they in fact needed the new construct only on rare occasions, thus suggesting a structured programming discipline was followed most of the time in those programs. Such results suggest that the inaccuracy introduced by a technique that would not correctly represent the above break statements would only occur on rare instances. In our work, we therefore have a working assumption, which is that a structured programming discipline was followed when coding the reverse-engineered program. Future work will look into relaxing this assumption.[7]

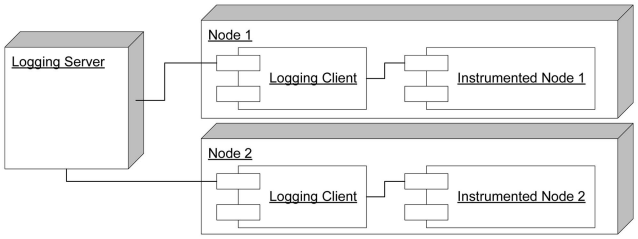

Fig. 11. Instrumentation architecture (deployment diagram).

The SUS is instrumented such that instrumentation statements (inserted using aspects) interact with class `LoggingClient`. Each node of the instrumented distributed SUS has one instance of this class (a thread safe implementation [36] of the singleton design pattern [14]) which communicates, either at runtime or offline, with the trace repository called `LoggingServer` (Fig. 11). This added node to the distributed SUS collects the trace statements, joins them together (e.g., recognizing remote calls), and performs the necessary transformations, to generate scenario diagrams. These transformations are performed offline to obtain complete traces and thus the impact on the behavior of the instrumented SUS is minimized. Last, once the user specifies which classes have to be monitored, the generation of the aspects is automated.

In this section, we first describe what information we collect at runtime so as to uniquely identify instances in the SUS and uniquely identify the events we observe such as the start of a method execution (Section 4.1). We then present the different aspects that we have defined to retrieve information on program execution: Execution of methods and constructors (Section 4.2), RMI communication (Section 4.3), and control-flow structure within methods (Section 4.4). Notice that, due to space constraints, Section 4.2 only reports on tracing constructor execution (tracing method executions being very similar) and Section 4.4 only reports on tracing control flow on a representative example, i.e., `if` statements. Instead of showing actual aspects that would be specific to a particular SUS, we show aspect *templates* that present the general structure of the aspects, highlighting the variable (system dependent) parts in bold face.

4.1 Unique Identifiers

In order to identify messages exchanged by class instances in the (distributed) SUS, we must first be able to uniquely identify class instances in the SUS. To do so, we give two unique identifiers (namely, `objectID` and `nodeID`) to instances of classes whose behavior has to be monitored. This is done by the aspect in Fig. 12. AspectJ allows aspects to add attributes (lines 2-4) and methods (lines 6-11) to existing classes. Additionally, the aspect adds an interface to the set of interfaces already implemented by the instrumented classes, namely `ObjectID` (AspectJ keyword `declare`). This will allow other aspects to cast class

7. Note that a similar issue relates to the use of `continue` and `return` statements. In particular, the authors of [33] suggest an extension to UML 2.0 to accurately represent return statements and reported that, in the 20 programs, such a notation was often needed. Addressing these cases will be the purpose of future work.

```
1  public aspect InstanceIdentifiers {
2    public int ClassName.objectID = ClassName.objectIDgenerator();
3    public int ClassName.nodeID;
4    private static int ClassName.currentObjectID = 0;
5
6    private static int ClassName.objectIDgenerator() {
7      return ClassName.currentObjectID++;
8    }
9    public int ClassName.getObjectID() { return objectID; }
10   public int ClassName.getNodeID() { return nodeID; }
11   public void ClassName.setNodeID(int id) { nodeID = id; }
12
13   declare parents : ClassName implements ObjectID;
14 }
```

```
1  public interface ObjectID {
2    public int getObjectID();
3    public int getNodeID();
4    public void setNodeID(int id);
5  }
```

Fig. 12. Adding object identifiers.

```
1  public class LoggingClient {
2    public synchronized static LoggingClient getLoggingClient() {…}
3    public synchronized void setNodeID(int node) {…}
4    public synchronized int getNodeID() {…}
5    public synchronized int instrument(List rec, Object[] args) {…}
6  }
```

Fig. 13. Definition of class LoggingClient (excerpt).

instances into `ObjectID` instances and call `getObjectID()` and `getNodeID()` to get the `objectID` and `nodeID` attribute values, respectively. The aspect in Fig. 12 shows that determining `objectID` is local to classes (static attribute `currentObjectID`) and, thus, local to nodes in the SUS. As a result, the pair (`objectID`, `className`) uniquely identifies an instance on a node in the SUS, but two instances of the same class on different nodes in the SUS can have the same `objectID`. Pair (`objectID`, `className`) does not uniquely identify instances in the SUS but, given that our instrumentation records a unique identifier for each node in the SUS (namely `nodeID`), triplet (`objectID`, `className`, `nodeID`) does. We will see in Section 4.2 that attribute `nodeID` is set by the aspects intercepting constructor executions.

The aspects discussed in the following sections all use class `LoggingClient` to report traces to the trace repository: Fig. 13 shows an excerpt of the class definition. One instance of this class is created on each node of the SUS network (synchronized singleton [14], [36]). Each instance stores an identifier `nodeID` to uniquely identify the instrumented nodes.

To report to the repository (`LoggingServer` in Fig. 11) on an event being observed, our aspects use method `instrument()` in class `LoggingClient`. The first parameter of this method is a `List` of `String` instances used by our aspects to provide details on the observed event. Its first element is always a string characterizing the observed event, which is used by the repository to recognize the different kinds of observed events: e.g., "`Create method start`" in the case of the start of a constructor execution. The remaining `String` elements are specific to the observed event, e.g., when observing the start of a constructor execution, the class name of the instance being created and the signature of the constructor are included in the `List`. The second parameter of method `instrument()` is used when tracing constructor and method executions to report on the argument values being used.

Method `instrument()` also reports on the `nodeID` uniquely identifying the node on which the event is observed as well as a `timestamp` to record when the observed event occurs. This `timestamp` is also returned by `instrument()`. The strategy we adopted to record those events is to simply increment an integer in a method of class `LoggingClient` (not shown in Fig. 13), as we only need to know whether an event occurred before or after another event (the elapsed time between the two events does not matter in our context).[8] If two events corresponding to executions in the same node have timestamps `x` and `y` such that $x < y$, then this means that the latter occurred after the former, and then the trace repository can order the two events. We thus do not require calls to `System.CurrentTimeMillis()` or `System.nanoTime()`, or a global clock for the distributed system. An important consequence of our choice is that we need to use the pair (`nodeID`, `timestamp`) to uniquely identify an observed event (resulting in a trace statement) in the distributed SUS.

4.2 Method and Constructor Executions

We need to intercept any execution of any method or constructor in the SUS in order to instantiate class `LocalExecution` in the trace metamodel. Since different instrumentations are necessary for constructors and (static) methods, three different aspects are used. They are, however, very similar and we only describe the one for tracing constructor executions in this paper. We refer the reader to [6] for the two other aspects.

8. Note that other strategies are required to show execution times or durations in sequence diagrams.

```
1  Object around(): execution(PackageName..*.new(..))
2                   && !within(Instrumentation.*)       {
3    ArrayList log = new ArrayList();
4
5    log.add("Create method start");
6    log.add(thisJoinPointStaticPart.getSourceLocation().getWithinType()
7              .getName());
8    log.add(thisJoinPointStaticPart.getSignature().toLongString());
9    LoggingClient.getLoggingClient().instrument(log, thisJoinPoint.getArgs());
10
11   proceed();
12
13   ObjectID oID = ((ObjectID) thisJoinPoint.getThis());
14   oID.setNodeID(LoggingClient.getLoggingClient().getNodeID());
15   log.clear();
16   log.add("Create method end");
17   log.add(String.valueOf(oID.getObjectID()));
18   LoggingClient.getLoggingClient().instrument(log, null);
19   return null;
20 }
```

Fig. 14. Tracing constructor executions.

The aspect template in Fig. 14 specifies an around advice as we want to produce trace information (including timestamps) before and after the execution of the constructors. Indeed, we want to detect when constructors start and end executing in relation to other events as this will tell us what are their nested statements (Fig. 2): trace statements within the same node (i.e., same `nodeID`) between the two timestamps reported at constructor start and end. The point cut specifies any execution of method `new` (AspectJ notation for constructors) with any number of parameters (`new(..)`), on any class in a given package[9] (`PackageName..*`). Note that any around advice must return an instance of type `Object` (`Object around():...`), though it is not useful for constructors (the aspect simply returns `null`). The point cut also prevents any tracing of the aspects themselves, assuming aspect classes are in a specific `Instrumentation` package (keyword `within`).

Around the intercepted constructor execution (keyword `proceed`), the aspect prepares trace information (stored in the `log` variable) that is eventually sent to the trace repository, using method `instrument()`. Fig. 14 shows that the trace statement reporting on the start of a constructor execution contains a specific keyword specifying the intercepted behavior (string "`Create method start`"), the name of the class which constructor is executing (`..getName()`), the signature of the constructor (`...getSignature()...`) and the arguments used during the call to the constructor (`...getArgs()`): lines 5-9. Once the constructor has proceeded, `log` contains a specific keyword specifying the intercepted behavior (string "`Create method end`") and the object identifier of the newly created object: lines 13-18. (The object is cast to the interface `ObjectID` to be able to use the method `getObjectID()`.) The `nodeID` attribute of the newly created object is also set once the constructor has proceeded.

4.3 RMI Communications

Using RMI, a call to a remote method (i.e., a call to an object whose class implements `java.rmi.Remote`), is similar to a call to a local object. For instance, RMI provides synchronous communication: When the RMI client invokes a remote method at the server, it is blocked until the method invoked returns. If a similar aspect to the one presented above is used to monitor the execution of the client and the remote methods, the data generated (node, object identifiers, class name, and timestamps) is not sufficient to identify which client method called which remote method. We thus have to intercept executions of methods in classes implementing interface `java.rmi.Remote` on the server side (Section 4.3.2), and calls to methods on objects which class implements interface `java.rmi.Remote` on the client side (Section 4.3.1). Our strategy is to include in the trace statement for the remote method execution data on the client trace statement. To do so, some information has to be passed along the calls to remote methods.

4.3.1 Server Side

The aspect template in Fig. 15 wraps around methods in `java.rmi.Remote` interfaces on the server side. For each such remote method, named, for instance, `MethodName(...)`, the aspect adds method[10] `MethodNameExtra(...)` (with its body) to the interface.[11] This aspect is different from the previous ones as no pointcut is defined: We do not intercept anything but only add a method to an interface. This added method has two more parameters than the original one, both of type `int`, which are used by the client performing the call to pass information about itself: pair (`nodeID`, `timestamp`) that uniquely identifies the trace statement corresponding to the call on the client side (recall the attributes of class `RemoteMethodExecution` in Fig. 2). `MethodNameExtra(...)` thus 1) uses the added parameter containing client information to produce the trace on the server side (lines 9-14), 2) calls the original method `MethodName(...)` with the original arguments[12] (line 16), and 3) produces a new trace statement before

9. The following specifies multiple packages or classes: `( execution(Package1..*.new(..)) | | execution(Package2..*.new(..)) | | execution(Package3.className.new(..)) )`.

10. This assumes that the interface does not already have a `MethodNameExtra()` method. This can be solved easily by adding unique, automatically generated strings to `MethodName` until we find a name that is not already part of the interface.

11. AspectJ allows the addition of concrete methods with bodies to interfaces, though this is not allowed in standard Java.

12. The execution of the original method will be intercepted by the aspects discussed in Section 4.2.

```
1  public Object PackageName.InterfaceName.MethodNameExtra(
2          int clientNodeID,
3          int clientTimeStamp;
4          [, parameters of the method, if any]
5          ) throws RemoteException
6          [, other throws clauses, if any]      {
7    ArrayList log = new ArrayList();
8
9    log.add("Remote method execution start");
10   log.add(String.valueOf(((ObjectID) this).getObjectID()));
11   log.add(this.getClass().getName());
12   log.add(String.valueOf(clientNodeID));
13   log.add(String.valueOf(clientTimeStamp));
14   LoggingClient.getLoggingClient().instrument(log, null);
15
16   Object ret = MethodName([parameters of the method, if any]);
17
18   log.clear();
19   log.add("Remote method execution end");
20   LoggingClient.getLoggingClient().instrument(log, null);
21   return ret;
22 }
```

Fig. 15. Tracing remote method executions.

returning the result from the original method. This way, traces on the server side contain enough information to uniquely match trace statements on the client side (call) with trace statements on the server side (execution). This results in a caller-callee relationship between an instance of class `RemoteMethodExecution` (corresponding to the execution of the added method `MethodNameExtra`) and an instance of `LocalExecution` (corresponding to the execution of the original method `MethodName`). The aspect in Fig. 15 is for remote methods that have a return value. Another, very similar aspect for methods without any return value can be found in [6].

4.3.2 Client Side

On the client side, any call to a remote method must also be intercepted to instead perform a call to "Extra" remote methods and add the required first two parameters (client information). This is the purpose of the template in Fig. 16. This is an around advice intercepting calls to remote methods (line 3). The signature also allows the advice to access the object on which the call is performed using reference name `targetObj` (line 5). The around advice is necessary as we have to change the call, and only an around advice permits it. (Note that this aspect template is for intercepting calls to nonstatic remote methods: Another similar template for static remote methods can be found in [6].)

The advice first tests whether the class name of the object on which the call is performed ends with string "`_Stub`" (line 8), i.e., whether the object is an RMI stub representing a remote object.[13] This is necessary as the point cut also specifies calls, local to the server, to methods in `Remote` interfaces. These are local calls to the server and should not result in trace statements indicating remote calls: In such a case, the advice does nothing but to proceed with the execution, without any trace statement produced (line 21).

13. This assumes that the stub was created by the RMI compiler `rmic` or was named "`..._Stub`" if it was user defined. Future work will address this limitation, though we do not foresee any difficulty.

The aspect then produces a trace statement on the client side (lines 9-11): A "`Remote method call start`" with a specific `timestamp` and `nodeID`. Note that no context (e.g., `objectID` and/or `classname`) is reported in the trace as we assume the context of a `RemoteCall` instance is the same as its caller (Section 3.2). The aspect then performs the call to the extra method (lines 13-14), providing client trace data in the two first arguments (`nodeID` is retrieved from `LoggingClient`, and the `timestamp` is returned by the previous call to `instrument()`) along with possible other arguments of the original call. It finally produces a trace statement indicating the end of the execution of the extra method (lines 16-18).

As discussed in Section 4.1, pair (`nodeID`, `timestamp`) uniquely identifies a trace statement in the distributed SUS, i.e., two different trace statements in the SUS may have the same `timestamp` if reporting on executions on two different nodes, but they are uniquely identified when additionally accounting for the `nodeID`. Since our instrumentation of RMI calls/executions reports on this unique identification of the client trace statement in the server trace statement, it is possible to uniquely match trace statements from the client and server sides.

This unique matching of client and server traces also allows us to handle messages lost by the network. If this happens, a trace statement on the client side cannot be matched to any server side trace statement. This situation can be automatically detected and reported by the `LoggingServer` when it analyzes traces received from the different nodes in the SUS. Another mechanism to identify lost RMI requests would be to instrument try-catch blocks (on the client side) since the remote call would then throw an exception. Notice that our approach would also allow us to consider other distribution middleware than RMI, even if they do not entail synchronous communications.

Another issue worth mentioning relates to the RMI parameter passing mechanisms: Nonremote objects are

```
1  Object around(Object targetObj [, Types and parameter names if any])
2    throws RemoteException:
3      call(* PackageName..*.MethodName([Parameter types if any]) throws
4          RemoteException [,other throws clauses if any])
5      && target(targetObj)
6      && args([ParameterNames if any])
7      && !within(Instrumentation.*)      {
8    if (targetObj.getClass().getName().endsWith("_Stub")) {
9      ArrayList log = new ArrayList();
10     log.add("Remote method call start");
11     int timestamp = LoggingClient.getLoggingClient().instrument(log, null);
12
13     Object ret = targetObj.MethodNameExtra(LoggingClient.getLoggingClient()
14         .getNodeID(),timestamp [, arguments of the original method if any] );
15
16     log.clear();
17     log.add("Remote method call end");
18     LoggingClient.getLoggingClient().instrument(log, null);
19     return ret;   // the result of MethodName's execution
20   } else {
21     return proceed(targetObj);
22 }
```

Fig. 16. Tracing calls to (nonstatic) remote methods.

passed by copy, whereas remote objects are passed by reference. When a nonremote object is passed by copy, the SUS contains two copies of the same data, and this can lead to inconsistencies if the server modifies its copy without notifying the client. However, passing objects by copy improves performance since the server does not have to use the network to retrieve information (e.g., an attribute value) about the object. On the server side, two different kinds of calls can be performed on the copy of the client object: calls that only query the object state and calls that modify the object state. We consider the latter an erroneous use of RMI that can likely result in inconsistent data in the SUS: If both the client and the server nodes need to modify the object state, the object should be passed by reference, not by copy. Following our instrumentation strategy, passing an instrumented class instance by copy results in two objects, one on the client side and one on the server side, that have the same triplet (`objectID`, `className`, `nodeID`): the `objectID` and `nodeID` are attributes of the object that have been determined on the client side (e.g., the `nodeID` is the unique identifier of the client node). In other words, from the point of view of tracing executions on objects in the SUS and abstracting the messages they exchange, these two objects actually represent one and only one object: calls to methods on these two objects will result in messages with the same target classifier.

4.4 Control-Flow Structures

As stated earlier, we want to provide enough information in traces so that scenario diagrams can be generated. At the same time, we want to represent object interactions using the UML 2.0 notation. This includes repetitions of messages and conditions under which messages are exchanged by objects. Contrary to existing techniques that aim at discovering patterns of executions in traces (Section 2), we opted for the instrumentation of control flow structures. However, AspectJ does not currently provide mechanisms to intercept method control flow and, as a temporary measure, we have to resort to instrumenting the SUS source code. In order to keep this instrumentation minimal, we defined a class within the aspect code (i.e., in package `Instrumentation`), namely, `TracingCTRLFlow`, that provides operations (with empty bodies) for most of the Java control flow structures.[14] For instance, it has methods `ifStatementStart(String, String)` and `ifStatementEnd()` for the start and the end of an `if` statement. (The two parameters of the former method are used to report on the complete if statement, i.e., the complete string `if(...)`, and the expression being evaluated in the `if` statement, respectively.) Calls to `TracingCTRLFlow` methods are inserted in the SUS source code at appropriate places to be intercepted for instrumentation purposes: at the beginning and end of `while`, `do`, `for`, `if`, `else`, and `case`[15] blocks, as well as immediately before `break`, `continue`, and `return` statements. Note this is only a temporary measure as intercepting method control flow has been identified as a possible addition to future releases of AspectJ. (This has been discussed on mailing lists by the developers of AspectJ.) An alternative solution is to use a static analysis technique to retrieve the control flow graph and then to devise a procedure (similar to pattern matching) to compare the control flow graph, which captures all the possible control flow paths, with the control flow paths actually detected from traces.

Calls to methods in `TracingCTRLFlow` are then intercepted by specific aspects, reporting on the kind of control flow structure being executed, the context of the

14. Our instrumentation strategy does not currently support "?:" statements, or labeled break and continue statements. A "?:" statement can for instance be transformed (for tracing purposes only) into an if-then-else statement. Tracing labeled break and continue can report on the target location of the jump to facilitate reverse-engineering.

15. The calls are inserted before the first statement of each `case` (i.e., after character ":") and before the `break` of each case. Note that we thus assume there is a `break` statement at the end of each case and therefore our instrumentation strategy does not currently support fall-through semantics in `switch` statements. The methods being called are those used for an `if` block. The result is therefore an `alt` combined fragment in the generated sequence diagram.

```
1 before (String statement, String boolExpression):
2   call(public void Instrumentation.TracingCTRLFlow.ifStatementStart(..))
3   && args(statement, boolExpression) {
4     ArrayList log = new ArrayList();
5     log.add("If start");
6     log.add(statement);
7     log.add(boolExpression);
8     LoggingClient.getLoggingClient().instrument(log, null);
9 }
```

```
1 before() call(public void Instrumentation.TracingCTRLFlow.ifStatementEnd(..)){
2   ArrayList log = new ArrayList();
3   log.add("If end");
4   LoggingClient.getLoggingClient().instrument(log, null);
6 }
```

Fig. 17. Tracing the start and end of an if statement.

execution (objectID, className, nodeID, timestamp), and the clause (condition) driving the flow of control (plus the initialization and update parts of for loops).

Fig. 17, for instance, shows the two aspects for tracing the start and end of an if block (either the then or else parts of an if statement), i.e., the aspects for intercepting calls to methods ifStatementStart(...) and ifStatementEnd(...) in class TracingCTRLFlow. The first aspect generates a trace labeled "If start" and reports on the complete if statement and its condition. The trace generated by the second aspect is simply labeled "If end." A complete list of aspects intercepting control flow structure executions can be found in [6].

This solution minimizes the instrumentation of the source code as aspects are responsible for producing traces, as opposed to additional statements in the source code.

The logging server is able to recognize alternative or repetition trace statements thanks to the specific trace statements reported at the beginning and end of control flow structures: e.g., beginning and end of the else part of an if statement, beginning and end of a while loop. This process is disrupted by break, continue, and return statements. For instance, when a break executes in an if block, itself in a while loop, the trace shows the start of the while, the start of the if, and the execution of the break (a call to method breakStatement() in class TracingCTRLFlow is intercepted) but does not contain a trace reporting on the end of the if or the end of the while. The statements added to the source code to be intercepted by our aspects do not get a chance to execute. This is, however, not a problem since using the trace statement reporting on the execution of break, the logging server is able to complete the trace after it is generated: it is able to recognize that the end of the if and the end of the while occurred. The same principle applies when continue and return are used in control flow structures.

Although our approach does not currently trace Java exception handling mechanisms, we consider using a similar instrumentation strategy (i.e., instrumenting the source code with calls to startCatch() and endCatch() in class TracingCTRLFlow). We do not foresee specific problems since, for our tracing purpose, a thrown exception disrupts the control flow in a way similar to break, continue and return: The logging server should be able to complete traces, accounting for the end of control flow structures but also method calls. Note that AspectJ allows the definition of a pointcut for exception handlers (i.e., catch blocks). Unfortunately, the only advice currently available for this kind of pointcut is a before advice, while we not only need to trace the start of the handler but also its end in order to be able to distinguish the control flow within the handler from the control flow after the try-catch structure.

5 Case Study

We selected a Library system as a case study, since it provides a complete set of functionalities (adding customers, titles, copies, making reservations...) and proper Analysis and Design documents (including sequence diagrams) were designed under the authors' supervision. The size of the system, excluding comments and blank spaces, is approximately 7,000 LOCs. Furthermore, it was developed independently from the current work, in the framework of a fourth-year engineering project. The system is not only implemented in Java, but is distributed (several client nodes can communicate with a unique server node), and the middleware for the network communications is RMI.

We then find ourselves in a typical context where the reverse engineering of sequence diagrams can be useful to check the consistency of the code with the design. We carefully compare reverse-engineered and design sequence diagrams and determine the cause of discrepancies, whether they are due to inconsistencies, implementation decisions, or mistakes in our algorithms.

All the classes in the Library system have been instrumented using the strategy described in Section 4. Two exceptions are classes belonging to the Graphical User Interface (GUI) and classes implementing a database management system (for storing instances of entity classes Copy, Title, ...). The reason is that we are not interested in the reverse-engineering of those subsystems as their object interactions are usually not described in the Analysis/Design sequence diagrams, which focus on interactions among application objects. GUI objects interact through boundary classes [8] and the DBMS is accessed using a Façade [14].

We have selected the "Add copy" use case as a representative example, as it illustrates the most important aspects of the reverse-engineering process: conditional messages, RMI remote calls, and object creations. Though this case study may appear of limited size, recall that our

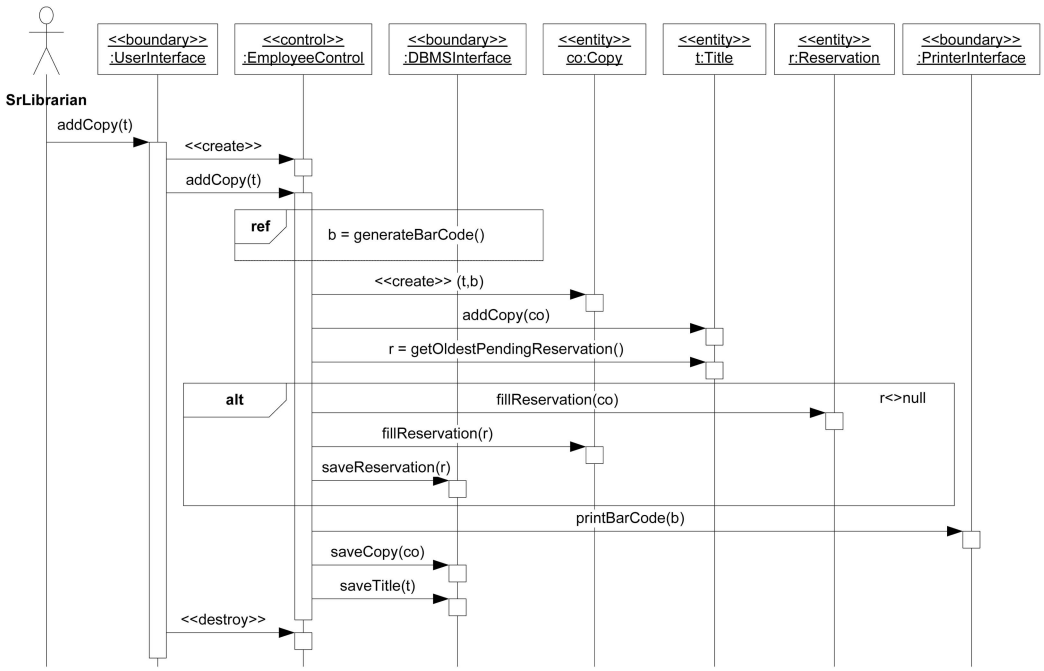

Fig. 18. Sequence diagram provided in Analysis and Design documents.

Client side – nodeID = 1

```
Method start*11*1*0*Employee.EmployeeControlIFFacade*public long
  Employee.EmployeeControlIFFacade.addCopy(java.lang.String)*addCopy*?java.lang.
  String?123-4567?
Remote method call start*12*1
Remote method call end*13*1
Method end*14*1*?java.lang.Long?1?
```

Server Side – nodeID = 0

```
Remote method execution start*55*0*server.EmployeeControl*12*1
Method start*56*0*0*server.EmployeeControl*public long
  server.EmployeeControl.addCopy(java.lang.String)*addCopy*?java.lang.String?123
  -4567?
…
Remote method execution end*136*0
```

Fig. 19. Excerpt of the trace for use case "Add copy."

focus is on retrieving relevant and complete information on object interactions from dynamic analysis rather than to address the visualization problem for large systems.

When the "Add copy" use case is triggered, the Analysis/Design documentation (Fig. 18) indicates that message `addCopy(Title)` is sent to an instance of class `EmployeeControl` (only employees can add copies). Adding a copy for a given title then first consists in generating a barcode which is eventually printed on a sticker (the interaction occurrence `ref` specifies that the corresponding interaction can be found in another sequence diagram), creating a copy object (with the `Title` object and the `BarCode` object passed to the constructor of `Copy`), and adding the copy to the list of copies held by the title. Now that a new copy has been added for the title, pending reservations on the title, if any, can be dealt with: This is the purpose of the `alt` combined fragment. Last, the new copy and the title objects can be saved into the database.

We have executed use case "Add copy" on the instrumented version of the Library. This was performed on a PC running Windows XP with a Pentium 4 processor at 2 GHz and 1 Gb RAM. We used JDK 1.5 and AspectJ 1.2.1. The generated traces (client and server sides) have been used to produce an instance of the trace metamodel, and an instance of the scenario diagram metamodel has been generated according to the consistency rules described in Section 3.3. The trace size, accounting for both the client and server sides, was of the order of 40 KB. Fig. 19 is an excerpt of the traces for the "Add copy" use case (client and server sides), showing only trace statements related to RMI, and complete traces can be found in [6]. Fig. 19 shows a call to a remote method on the client and a matching remote method execution on the server side: The trace at the client side reports on a method execution on an instance of class `Employee.EmployeeControlIFacade(timestamp=11, nodeId=1,objectID=0)`, and a remote call (`timestamp=12, nodeID=1`), and the trace at the server side reports on a remote method execution (`clientTimestamp=12, clientNodeID=1`).

Fig. 20 shows the scenario diagram[16] corresponding to the trace [6]. This illustrates the usefulness of the approach as discrepancies between this figure (reverse-engineered scenario diagram) and Fig. 18 (sequence diagram produced during Design) are clearly visible. Not having in Fig. 20

16. Note that when reporting arguments used during calls, we show the value of primitive types and class name and object identifiers (`objectID`) of user defined types.

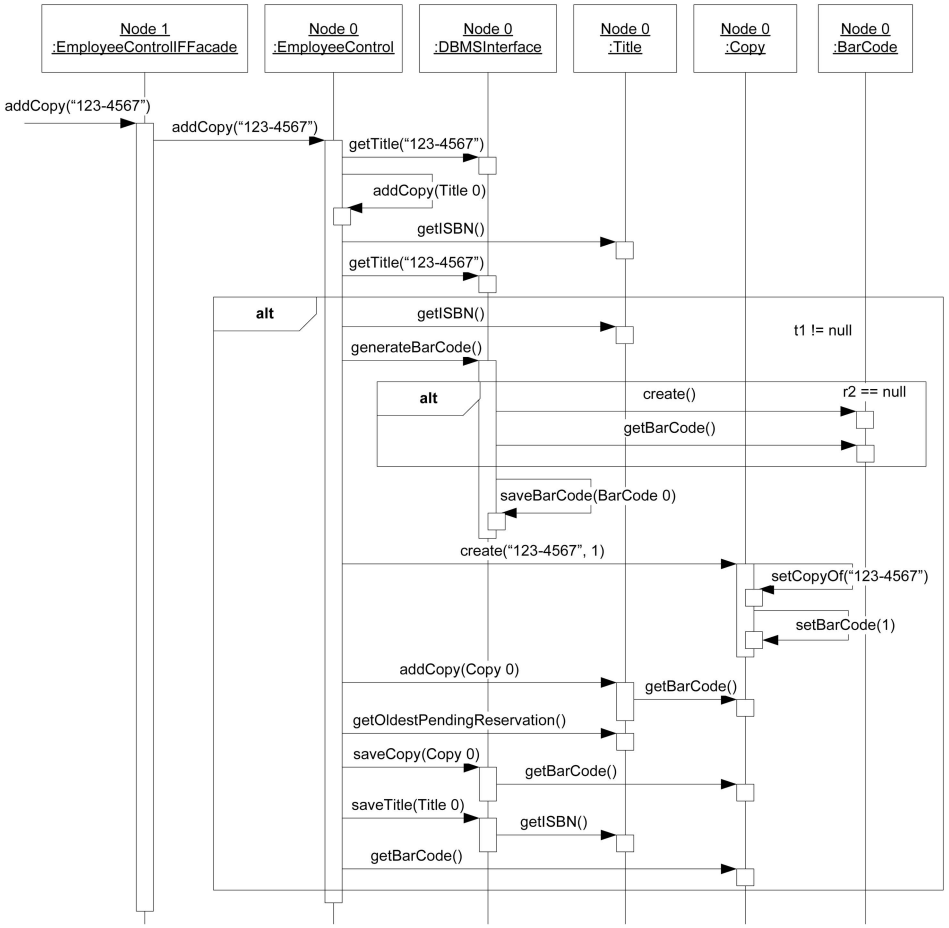

Fig. 20. Scenario diagram produced from the trace in [6].

counterparts for messages in the `alt` combined fragment in Fig. 18 is not an error as we have added a copy to a title that does not have reservations (`getOldestPendingReservation()` returns `null`): Fig. 20 is only a possible scenario. Similarly, not having in Fig. 18 a counterpart for the messages triggered by `generateBarCode()` in Fig. 20 (e.g., the nested `alt` combined fragment) is not an error: The reverse-engineered scenario diagram simply shows the details of the interaction occurrence labeled `ref` in Fig. 18.

Discrepancies can be found in parameter types: `addCopy()` has a parameter of type `Title` in Fig. 18, whereas it has a parameter of type `String` in Fig. 20 (the ISBN). Similarly, `Copy`'s constructor has two parameters of types `Title` and `BarCode` in Fig. 18 (parameters `t` and `b`) and two parameters of types `String` (the ISBN of the corresponding title) and `int` (the barcode number) in Fig. 20. Since the parameter to `addCopy()` is a `String` instead of a `Title`, there is then a need for message `getTitle()`, which is sent to the database to return a reference to a `Title` object.

Fig. 20 also indicates that the developers decided to store barcode and ISBN numbers as opposed to `Copy` and `Title` references implementing the association between class `Title` and class `Copy` (e.g., when saving a `Title`, `DBMSInterface` asks for its ISBN). When creating a `Copy` object, only the ISBN of the title is passed as a parameter as the constructor does not require a reference to the `Title` object. When adding a copy to the title (message `addCopy(Copy 0)`), the `Copy` object is passed as a parameter, but the method asks the `Copy` object its barcode (if the title was storing the `Copy` reference, it would not need the barcode). This example illustrates how using reverse-engineered scenario diagrams can inform us about implementation choices.

Last, Fig. 20 shows unnecessary calls to `getISBN()` and `getBarCode()` at the beginning and end of the `alt` combined fragment, respectively (after looking at the source code, these calls could have been avoided by using local variables). This illustrates how the reverse-engineering of scenario diagrams and their comparison to design sequence diagrams can help improve the implementation. In other words, it can be used as a quality assurance mechanism. Future work will attempt to automate this comparison process.

In order to assess precisely the sources of the execution overhead introduced by our instrumentation in the "Add copy" scenario described above, we investigate separately the overhead introduced by RMI calls alone, by the control flow alone, and the complete instrumentation including RMI, control flow and local calls. To do so, for each type of instrumentation, we run the scenario 100 times to account for possible random variations due to transient changes in the OS background tasks. To limit other possible random

TABLE 3
Execution Elapsed Times and Overheads

Instrumentation	Duration in nanosec. (average)	Overhead due to (absolute/relative)	
None	16200305	RMI	4326046 / 27%
Full	38681198	Control flow	1966602 / 12%
Only control flow	18166907	Local calls	16188246 / 100%
Only RMI	20526351		

variations due to the environment, both the client and server are running on the same computer, no other application was running, and the computer was not connected to any network. These would introduce random delays (that we are not interested in) that would only introduce further noise in our analysis. Each scenario start and end times were computed using `java.lang.System.nanoTime()` (JDK 1.5), which returns the current value of the most precise available system timer, in nanoseconds. Appropriate delays were introduced between the executions of scenarios in order to avoid the overloading of the server over time, which would otherwise introduce unwanted variations in our results.

The results summarized in Table 3 should be interpreted by keeping in mind that, as reported in the trace, the "Add copy" scenario presented above includes one intercepted RMI remote method call (client) and method execution (server), 13 intercepted local method executions, two constructor executions, six static method executions, and two intercepted control flow constructs. Due to the larger number of local method executions (i.e., static and nonstatic methods, and constructors), this is clearly the main source of overhead (100 percent). Indeed, with our instrumentation strategy, every method execution involved during the "Add copy" scenario is intercepted. In second position comes the instrumentation of the intercepted remote method call and execution, with a relative overhead of 27 percent. For each RMI call and execution, our instrumentation entails

1. the interception of the call on the client side,
2. the additional call to the extra method,
3. the extra execution on the server side, and
4. a larger stub (produced by the RMI compiler `rmic`, after weaving) leading to more loading time (the stub class file size almost doubles).

Reductions of this overhead can be investigated, as discussed below. The intercepted control flow comes last with a relative overhead of 12 percent. This basically corresponds to the overhead of intercepting calls with AspectJ since our advices do nothing but to produce a trace statement. Note that when comparing the various elapsed time distributions for different instrumentations, statistical testing tells us that the overheads created by intercepting RMI calls, local calls, and control flow constructs are all statistically significant despite the random variation present in the execution time data.

More generally, the overhead incurred for different scenarios will approximately be proportional to intercepted events (RMI call and executions, local executions, control flow interception). In the case of RMI, for example, we wrap and add two parameters of type `int` to each remote method execution (Section 4.3.1). The resulting execution overhead will then be incurred for each intercepted remote method execution during the scenario execution. The overall overhead will therefore be proportional to the number of intercepted remote method executions, but the relative overhead will depend on the actual parameters of remote methods: the relative overhead due to our two extra parameters will decrease as the number of actual parameters increases. It is clear from the above results that our instrumentation introduces a significant overhead that will visibly slow down execution. However, it is not expected that it would prevent its usage for the purpose of reverse engineering sequence diagrams in most situations, except perhaps in the context of hard real-time systems with deadlines. Furthermore, in a well-designed distributed system, the usage of remote calls should be minimized to obtain acceptable performance, thus significantly decreasing the main source of overhead in our instrumentation.

Similar overhead results were obtained for other scenarios in the Library system, showing execution overheads per intercepted event within similar ranges, thus suggesting the results presented in Table 3 are representative. Given our instrumentation strategy this is not really surprising as the overhead incurred by intercepting local executions, remote calls and executions, and control flow constructs is relatively constant: The bodies of our advices do not depend on the intercepted functionality.

Considering recent studies on the dynamic behavior of AspectJ programs [11], we are considering changes to our aspects to improve efficiency. For instance, around advices are known to be more expensive than a before or after advice. Our around advice for intercepting calls to RMI remote methods on the client side could easily be replaced by a before advice: Since no further executions (i.e., `nestedStatements`) are induced on the client side, we do not need to report on both the start and end of those calls. Alternatively, recent optimizations of the AspectJ compiler (in particular, for the around advice) are encouraging: for example, the work in [1] suggests an optimization of the around advice that leads to more compact code and can also improve runtime performance. As discussed in [11], too generic pointcuts (e.g., `execution(PackageName..*.new(..))`) also cause significant overhead. Another possible area for improvements might be to

consider variations of AspectJ to define pointcuts that are specific to RMI [26].

6 Conclusion

This article provides a methodology to reverse engineer scenario diagrams—partial sequence diagrams for specific use case scenarios—from dynamic analysis. It does so by accounting for issues related to distribution. Though our solution is specific to the Java/RMI context, many of its components can be reused and tailored to other platforms.

A more general, methodological contribution of our work is the way we specified our reverse-engineering process by using metamodels (as UML class diagrams) and transformation rules (as constraints in the Object Constraint Language). Our review of the literature has shown that many reported works were not described in sufficient details and formality so as to allow formal comparisons and improvements. Our approach enables the specification of what information traces contain at a logical level and its mapping to a formal, abstract model (in our case, a scenario diagram). Future work can then be compared by comparing metamodels and mapping rules. Another advantage is that our metamodels and rules can then naturally be used as specifications to develop tool prototypes. The initial class structure of our prototype was indeed a direct reflection of our metamodels. It is also important to note that if our reverse engineering process were to be adapted to a different distribution middleware, the metamodels and rules would remain unchanged, except that timestamps might be measured in a different way (e.g., according to a global clock instead of local clocks in the case of asynchronous remote communications). Only the instrumentation would then be affected.

In order to minimize the impact of source code instrumentation and its related drawbacks, we used Aspect-Oriented Programming (AOP) to instrument the bytecode of Java systems. This brings a lot of benefits both in terms of the overhead and practicality of instrumentation and enables the clear separation of instrumentation and application code. We provide here a set of precise aspects, including some that deal with distribution issues in the context of Java/RMI. If a different middleware were to be used, the aspect code might have to measure time in a different way (e.g., global clock) and all time-related statements would change. Also, all the aspect code statements referring to the `java.rmi.remote` interface would change to refer to whatever interface is defined by the new middleware to interact with remote objects. A change of language would, of course, have a much more serious effect as a different aspect weaver would have to be used, possibly following a very different syntax.

We performed a case study on a distributed library management system developed in Java, using RMI as distribution middleware. Our case study, though limited in size, allowed us to validate our metamodels and algorithms. It was also useful to illustrate how reverse-engineered scenario diagrams can be used to help check the quality of the implementation, its conformance to the design, and inform us about implementation choices. A careful study of the execution times in this system has shown that the overhead introduced by the instrumentation is significant but should not prevent the reverse engineering of scenario diagrams in most practical situations. Detailed results were reported for a representative scenario in terms of absolute and relative overheads due to instrumenting RMI remote calls, local calls, and control flow constructs.

Future work includes the automated derivation of sequence diagrams (i.e., *merging* of scenario diagrams), the automated comparison of complete sequence diagrams for testing and quality assurance purposes, and the use of a static analysis technique to capture the control flow inside methods more accurately than through instrumentation.

References

[1] P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Optimising aspectJ," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 117-128, 2005.

[2] N. Bawa, S. Ghosh, "Visualizing Interactions in Distributed Java Applications," *Proc. IEEE Int'l Workshop Program Comprehension,* pp. 292-293, 2003.

[3] B. Beizer, *Software Testing Techniques,* second ed. Van Nostrand Reinhold, 1990.

[4] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide,* second ed. Addison-Wesley, 2005.

[5] Borland: Together, http://www.borland.com/together, 2003.

[6] L.C. Briand, Y. Labiche, and J. Leduc, "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multithreaded Java Software," Technical Report SCE-04-04, Carleton Univ., http://www.sce.carleton.ca/Squall, Sept. 2004.

[7] L.C. Briand, Y. Labiche, and Y. Miao, "Towards the Reverse Engineering of UML Sequence Diagrams," *Proc. IEEE Working Conf. Reverse Eng.,* pp. 57-66, 2003.

[8] B. Bruegge and A.H. Dutoit, *Object-Oriented Software Engineering—Conquering Complex and Changing Systems.* Prentice Hall, 2000.

[9] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the Execution of Java Programs," *Proc. Software Visualization,* pp. 151-162, 2002.

[10] G.A. Di Lucca, A.R. Fasolino, P. Tramontana, U. De Carlini, "Abstracting Business Level UML Diagrams from Web Applications," *Proc. IEEE Int'l Workshop Web Site Evolution,* pp. 12-19, 2003.

[11] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge, "Measuring the Dynamic Behavior of AspectJ Programs," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 150-169, 2004.

[12] T. Elrad, R.E. Filman, and A. Bader, "Aspect-Oriented Programming: Introduction," *Comm. ACM,* vol. 44, no. 10, pp. 29-32, 2001.

[13] H.-E. Eriksson and M. Penker, *UML Toolkit*. Wiley, 1998.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[15] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering.* Prentice Hall Int'l Ed., 1991.

[16] J.D. Gradecki and N. Lesiecki, *Mastering AspectJ—Aspect-Oriented Programming in Java.* Wiley, 2003.

[17] Y.-G. Guéhéneuc and T. Ziadi, "Automated Reverse-Engineering of UML v2.0 Dynamic Models," *Proc. ECOOP Workshop Object-Oriented Reengineering,* 2005.

[18] R. Hofman and U. Hilgers, "Theory and Tool for Estimating Global Time in Parallel and Distributed Systems," *Proc. IEEE Euromicro Workshop Parallel and Distributed Processing,* pp. 173-179, 1998.

[19] IBM:Rational Test Real-Time, http://www-306.ibm.com/software/awdtools/test/realtime/, 2005.

[20] D.F. Jerding, J.T. Stasko, and T. Ball, "Visualizing Interactions in Program Executions," *Proc. ACM Int'l Conf. Software Eng. (ICSE),* pp. 360-370, 1997.

[21] R. Kollmann and M. Gogolla, "Capturing Dynamic Program Behavior with UML Collaboration Diagrams," *Proc. IEEE European Conf. Software Maintenance and Reeng.,* pp. 58-67, 2001.

[22] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf, "A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering," *Proc. IEEE Working Conf. Reverse Eng.*, pp. 22-32, 2002.
[23] D. Kortenkamp, R. Simmons, T. Milam, and J.L. Fernandez, "A Suite of Tools for Debugging Distributed Autonomous Systems," *Formal Methods and Systems Design J.*, vol. 24, no. 2, pp. 157-188, 2004.
[24] D. Mills, "RFC 2030—Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI," http://www.faqs.org/rfcs/rfc2030.html, 2005.
[25] J. Moe and D.A. Carr, "Using Execution Trace Data to Improve Distributed Systems," *Software—Practice and Experience*, vol. 32, no. 9, pp. 889-906, 2002.
[26] M. Nishizawa, S. Chiba, and M. Tatsubori, "Remote Pointcut—A Language Construct for Distributed AOP," *Proc. ACM Int'l Aspect Oriented Software Development*, pp. 7-15, 2004.
[27] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)," *Software Visualization*, pp. 176-190, 2002.
[28] "UML 2.0 Superstructure Specification," Final Adopted Specification ptc/03-08-02, Object Management Group, 2003.
[29] T. Pender, *UML Bible*. Wiley, 2003.
[30] M. Raynal and M. Signhal, "Logical Time: A Way to Capture Causality in Distributed Systems," technical report, IRISA, Jan. 1995.
[31] T. Richner and S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM)*, pp. 34-43, 2002.
[32] A. Rountev and B.H. Connell, "Object Naming Analysis for Reverse-Engineered Sequence Diagrams," *Proc. IEEE/ACM Int'l Conf. Software Eng.*, pp. 254-263, 2005.
[33] A. Rountev, O. Volgin, and M. Reddoch, "Static Control-Flow Analysis for Reverse Engineering UML Sequence Diagrams," *Proc. ACM Workshop Program Analysis for Software Tools and Eng.*, pp. 96-102, 2005.
[34] M. Salah and S. Mancoridis, "Toward an Environment for Comprehending Distributed Systems," *Proc. IEEE Working Conf. Reverse Eng.*, pp. 238-247, 2003.
[35] W. Schütz, *The Testability of Distributed Real-Time Systems*. Kluwer Academic, 1993.
[36] T. Sintes, "Singletons with Needles and Thread," http://www.javaworld.com/javaworld/javaqa/2002-01/02-qa-0125-singleton4.html, 2004.
[37] T. Systa, K. Koskimies, and H. Muller, "Shimba—An Environment for Reverse Engineering Java Software Systems," *Software—Practice and Experience*, vol. 31, no. 4, pp. 371-394, 2001.
[38] Y. Terashima, I. Imai, Y. Shimostuma, F. Sato, and T. Mizuno, "A Proposal of Monitoring and Testing for Distributed Object Oriented Systems," *Electronics and Comm. in Japan, Part 1 (Comm.)*, vol. 86, no. 10, pp. 33-44, 2003.
[39] P. Tonella and A. Potrich, "Reverse Engineering of the Interaction Diagrams from C++ Code," *Proc. Int'l Conf. Software Maintenance*, pp. 159-168, 2003.
[40] R.J. Walker, G.C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak, "Visualizing Dynamic Software System Information through High-Level Models," *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 271-283, 1998.
[41] J. Warmer and A. Kleppe, *The Object Constraint Language*, second ed. Addison Wesley, 2003.

Lionel C. Briand received the PhD degree in computer science, with high honors, from the University of Paris XI, France. He is with the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he is a full professor and has been granted the Canada Research Chair in Software Quality Engineering. He is also a visiting professor at the Simula Research Laboratory, Oslo, Norway. Before that, he was the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany. He also worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland. He has been on the program, steering, or organization committees of many international, IEEE and ACM conferences. He is the coeditor-in-chief of *Empirical Software Engineering* (Springer) and is a member of the editorial board of *Systems and Software Modeling* (Springer). He was on the board of the *IEEE Transactions on Software Engineering* from 2000 to 2004. His research interests include model-driven development, testing and quality assurance, and empirical software engineering. He is a senior member of the IEEE.

Yvan Labiche received the BSc degree in Computer System Engineering from the graduate school of engineering at CUST (Centre Universitaire des Science et Techniques, Clermont-Ferrand), France, a master's degree in fundamental computer science and production systems in 1995 (Université Blaise Pascal, Clermont Ferrand, France), and a PhD in software engineering in 2000 at LAAS/CNRS in Toulouse, France. While working toward his PhD, Yvan worked with Aerospatiale Matra Airbus (now EADS Airbus) on the definition of testing strategies for safety-critical, on-board software, developed using object-oriented technologies. In January 2001, Dr. Labiche joined the Department of Systems and Computer Engineering at Carleton University as an assistant professor. His research interests include object-oriented analysis and design, software testing in the context of object-oriented development, and technology evaluation. He is a member of the IEEE.

Johanne Leduc received the BASc degree in computer engineering from the University of Ottawa and the MASc degree from the Ottawa-Carleton institute of Electrical and Computer Engineering at Carleton University. She is currently working at Siemens Corporate Research in the software engineering department. Her research interests include model-based testing and code generation, quality assurance, and test process improvement.