Towards Reverse Engineering of UML Sequence Diagrams of Real-Time, Distributed Systems through Dynamic Analysis

By

Johanne Leduc

A thesis submitted to the Faculty of Graduate Studies and Research
In partial fulfillment of the requirements for the degree of
Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada

September 2004

Your file Votre référence
ISBN: 0-612-97416-2
Our file Notre référence
ISBN: 0-612-97416-2

PAGINATION ERROR.

TEXT COMPLETE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

ABSTRACT

This thesis proposes a comprehensive methodology and instrumentation infrastructure for 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. We formally define our approach using metamodels and consistency rules. The instrumentation is based on Aspect-Oriented Programming in order to alleviate the overhead usually associated with source code instrumentation. A case study is discussed to demonstrate the applicability of the approach on a concrete example.

ACKNOWLEDGEMENTS

My deepest thanks go to Dr. Briand and Dr. Labiche for all their guidance, help and support.

A special thanks to my husband Jason, without whom this would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

Software systems are getting increasingly large and complex. This makes systems difficult to design, implement and test. System complexity is not the only challenge. Ever changing requirements, cost, reuse and many other factors complicate these three tasks. The goal of software engineering is not only to decrease this complexity, but to better manage it. One way to accomplish this is through testing. The main goal of testing is to analyze a system to detect the differences between specified (required) and observed (existing) behaviour [7]. This work is a first step towards the comparison of system documentation, in the form of UML sequence diagrams, to the implementation.

The specific goal we are trying to reach is to reverse engineer UML sequence diagrams of a distributed and concurrent system using the information produced by executions of the system under study (SUS). The sequence diagrams generated could then be generalized and then compared to the original design document diagrams in order to find any discrepancies between the two. This may highlight behavioural differences between what the designers intended the system to accomplish and what the implementation actually carries out. Once at system level testing, testers could benefit greatly from the tool proposed in this work.

The Unified Modeling Language (UML) [40] provides a standard way of expressing both static and dynamic information in a graphical notation. Reverse engineering of UML class diagrams and of UML sequence diagrams is already available in some case tools [5, 27]. These tools use static information to generate the diagrams but, for the reasons

discussed in the next paragraphs, it would be desirable to reverse engineer a UML sequence diagram by using dynamic information. This manner of describing the system may be an important step in examining the interactions in a system.

The proposed objective is not a trivial task when considering that in distributed systems, there may be multiple clients requesting services from many servers. Many use cases may be executing at once and so must be separated, since sequence diagrams normally describe just one use case.

Another complexity involves concurrency. One can no longer assume that the hardware running the application has a single processor. Many threads of execution may be running at the same time. To further this concern, these processes may not be running on the same computer, but may run remotely. Distributed systems such as these are more and more common.

Obtaining dynamic information is yet another challenge for the proposed work. One way to observe a system is to analyze the design documentation or the code. This is called static information. Another way to observe a system is to collect a trace of the implementation during execution, called dynamic information. This type of information gives a direct view of how the implementation behaves. Unfortunately, collecting dynamic information is much more difficult than analyzing the code itself, but is necessary since using static information alone is insufficient: inheritance, polymorphism and dynamic binding may cause different methods to be called than expected when examining code.

Many object-oriented languages support inheritance, polymorphism and dynamic binding. These mechanisms can make static analysis quite complex. For example, Java makes use of subtype polymorphism [30]. This allows a reference to an object to point to an object of a different type, if this object is lower in its hierarchy. That is to say, a reference of type `Base` can point to an object of type `Derived`. If the derived object overrides a method, when this method is invoked this new code will be called regardless of the reference type. Though a variable is of type `Base`, the code from class `Derived` may be run when methods are overridden. This is just one example where run time behaviour may be too complex to analyze statically.

Above all, our goal is to reverse-engineer sequence diagrams using dynamic information. The system to reverse-engineer may be distributed; messages passed between nodes in a network must be included in the diagram. Also, the system may be concurrent. Many threads of executions may be running simultaneously and may communicate with each other. This must be shown in the diagram as well. There are additional goals we strive for. First, the impact of the instrumentation on the execution of the system must be minimal. As much as possible, the mechanism that produces trace statements must not create a *probe effect*, interference with the relative timing between processes [33], nor create an excessive overhead. Second, we want to avoid having inconsistent versions of the target code and of the instrumented code. The process of instrumenting the code should not be complicated nor time consuming, so that when there is a code change, the user can easily re-instrument the code. Ideally, the instrumentation mechanism would not modify the target code, but be completely separate. This is desirable since maintaining two versions of the same code can lead to inconsistencies and create maintenance

problems. Finally, we strive to formalize our approach in a clear and concise manner, but also be as general as possible to separate our principles from specific implementation details.

To verify that the proposed approach is successful in its goals, an implementation of a tool will be done. Though the design could be implemented in a manner that would accommodate many languages and communication mechanisms, this research will propose a solution for the Java language, using Java RMI as a means of communication between clients and servers. This limitation is proposed in order to demonstrate that the theory of this thesis is sound while not being laden with details of all possible implementation of distributed systems. The tool will then be applied to examples and a case study.

This thesis first presents a review of related works (Chapter 2). This includes research in reverse-engineering of distributed and of non-distributed systems, as well as commercial tools. The next chapters discuss the metamodel representations of scenario diagrams and of the trace. The former allows us to know what information is needed to construct a scenario diagram. The latter defines what information we expect from the trace, regardless of the instrumentation strategy. Because the trace is not of the same form needed for the scenario, we next define the transformation rules, expressed in OCL [42], that express the mapping between the two metamodels. Chapter 6 presents the instrumentation strategy and Chapter 7 presents the implementation of the trace processing (instantiating the metamodels). Next, a case study illustrates the approach.

CHAPTER 2 RELATED WORK

Many strategies aimed at reverse-engineering dynamic models, and in particular interaction diagrams (diagrams that show objects and the messages they exchange), are reported in literature. Two kinds of works are relevant to our approach: strategies aimed at reverse-engineering dynamic information for non distributed systems (section 2.1), and strategies targeting distributed systems (section 2.2).

2.1 Understanding Non-Distributed Systems

As for understanding non-distributed systems, differences between existing approaches are summarized in Table 1. Though not exhaustive, this table does illustrate the differences relevant to our work. The strategies reported in Table 1 [5, 6, 8, 15, 18, 24, 27, 29, 38, 41][1] are compared according to seven criteria:

> Whether the granularity of the analysis is at the class or object level. In the former case, it is not possible to distinguish the (possibly different) behaviors of different objects of the same class, i.e., in the generated diagram(s), class X is the source of all the calls performed by all the instances of X. In [6, 29], the memory addresses of objects are retrieved to uniquely identify them, though (symbolic) names are usually used in interaction diagrams. The reason is probably (this issue is not discussed in [29]) that retrieving memory addresses at runtime is simpler than using attribute names and/or formal parameter and local variable names to determine (symbolic) names that could be used as unique object identifiers: this

[1] [6] is a previous version of the current work.

requires more complex source code analysis (e.g., problems due to aliasing). Last, it seems that, in [29], 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. Source code analysis is not mentioned in [18] either. In the simple example they use, interacting objects can easily be identified as they correspond to attributes and as there is no aliasing. In [24] objects are identified by numbers, though nothing is said on how those numbers are determined. Last, [5] analyzes the source code and uses variable or attribute names to identify objects. This however is too simplistic as two different names in the source code can reference the same object.

The strategy used to retrieve dynamic information (source code instrumentation, instrumentation of a virtual machine, or the use of a customized debugger[2]), and the target language.

Whether or not the information used to build interaction diagrams contains data about the flow of control in methods, and whether the conditions corresponding to the flows of control actually executed are reported. Note that in [38], 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 [24].

[2] In the case of [18], this criterion is not applicable as the strategy only uses the source code and no execution trace is produced (no execution is required).

The technique used to identify patterns of execution, i.e., sequences of method calls that repeat in an execution trace. The authors in [24, 15, 29, 38] 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 patterns of execution that are detected by these techniques can distinguish the execution of loops from 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 [6] it is possible to identify repetition of messages due to loops since those programming language constructs are instrumented.

The model produced: Message Sequence Chart (MSC), Sequence Diagram (SD), Collaboration Diagram (CD). Note that in [18], since the control flow information is not retrieved and the approach only uses the source code, the sequences of messages that appear in the generated collaboration diagram can be incorrect, or even unfeasible. Also, the actual (dynamic) type of objects on which calls are performed, which may be different from the static one (due to polymorphism and dynamic binding), is not known. Note that such a static approach, though producing UML interaction diagrams with information on the control flow, is also proposed by tools such as Borland TogetherJ [5].

Table 1 – Related work for non-distributed systems

	Class vs. Object level	Information source	Language	Control flow	Conditions	Patterns	Models produced
[15]	Class	Source code instrumentation	C++	No	No	String matching (heuristics)	MSC
[41]	Class	Virtual machine	Smalltalk	No	No	No	Custom diagrams
[38]	Class	Customized debugger	Java	Yes	No	String matching	UML SD-like
[18]	Object	NA	Java	No	No	NA	UML CD
[29]	Object (memory address)	Source code instrumentation	Smalltalk	No	No	Provided by user	UML SD
[8]	Object	Virtual machine	Java	No	No	Recurrences of calls	UML SD-like
[24]	Object	Java debug interface	Java	No	No	No	UML SD
[5]	Object (source code names)	Source code parsing	Java	Yes	Yes	No	UML SD
[27]	Object	Source code instrumentation	C++, Ada, Java	No	No	No	UML SD
[6]	Object (memory address)	Source code instrumentation	C++	Yes	Yes	Loops	UML SD

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, as illustrated in Table 2. The five approaches discussed in this section [3, 20, 23, 32, 39] are compared according to six criteria (see Table 2). Note that none of these approaches provide information on the control flow (or conditions), and do not recognize repetitions of message sequences, as these aspects are not their main focus. The six criteria are:

Whether the granularity of the analysis is at the component or the object level. Note that we use the term component here, rather than class, since approaches for distributed systems tend to focus on remote calls between components and do not focus on inter-class communication, like some of the approaches 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 RMI, interceptors in CORBA) are inherently confined to providing information on components [3, 23, 39].

The strategy used to retrieve dynamic information: source code instrumentation [20], JVM profiler [32], data stream communications between distributed objects [3], CORBA interceptors that provide a hook at the remote procedure call level [23, 39]. Note that, though data stream communications between distributed objects are traced in [3], 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 ORB implementation. Last, in [20] 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.

The 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.

Whether the approach provides information on executing threads, and how it addresses 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 [20], time offsets between computers are calculated based on RFC 2030[3]. [23, 32] use techniques that have been proposed in the literature to capture causality between events of a distributed system [14, 28]. The other two approaches do not provide enough information with respect to the time issue: in [39] and [3], the authors use trace histories and timestamp respectively, and mention causal relationships between events but the descriptions lack details.

The model produced. Only two approaches provide UML sequence diagrams [3, 32]. (Note that in [32], 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

[3] This document describes the Simple Network Time Protocol (SNTP) Version 4, which is used to synchronize computer clocks in the Internet [22].

mechanisms to produce performance statistics [23] or check temporal constraints [20].

Table 2 – Related work for distributed systems

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

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, real-time 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, 11] and labeled with the signal name (i.e., threads communicate through signals);

(3) The control flow involved in the interactions (branches, loops), as well as the corresponding conditions.

None of the approaches in Table 1 and Table 2 covers all the above information pieces and the goal of the research reported in this thesis is to address issues (1) to (3) in a way which is the least intrusive possible for developers and testers. Another issue to tackle, 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 [6] and [18] in a non-distributed context and [32] in a distributed context, where metamodels are defined for traces. 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.

[4] Consistent with the UML standard [25], 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.

CHAPTER 3 SCENARIO DIAGRAMS

A sequence diagram is a UML model that shows object interactions; more specifically, the objects participating in an interaction and the sequence of messages exchanged, arranged in time sequence [31]. A sequence diagram is typically used to describe a use case and to identify the participating objects [7]. A sequence diagram can be used to describe either an *abstract sequence* (all possible interactions) or *concrete sequences* (one possible interaction or scenario, as referred to in this work) [7]. These are also known as generic and instance forms [34] respectively. In this work, we produce scenarios from the execution traces. Though not in the scope of this work, scenarios that describe the same use-cases could then be generalized to form abstract sequences.

The major difference between a sequence and a scenario diagram is that the former has branches that indicate multiple possible executions when different conditions are met. Also, repetitions are indicated by a box surrounding the messages that are repeated until the condition (shown at the bottom of the box) is fulfilled. Although not necessary, conditions and iterations are included in our scenario diagram, since these are easily extracted from the instrumentation traces, and show the circumstances in which the messages have occurred. However, iterations are not generalized as in sequence diagrams. Instead, a box around the messages is shown for every time the loop is repeated. For example, a loop that is executed three times and has three method calls inside, with the second and third call in an if-else construct would look like Figure 1 for a sequence diagram, but will appear like Figure 2a or Figure 2b in our scenario diagram. Figure 2 only shows two possible scenarios, though there are many more. In Figure 2a,

the guard condition is true in each execution of the loop therefore Message2 is sent, whereas in Figure 2b, the guard condition is false so Message3 is sent.

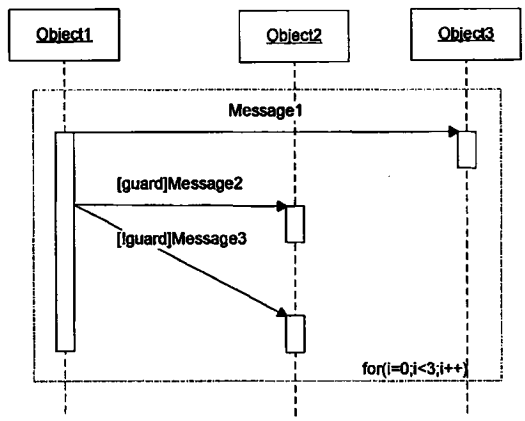
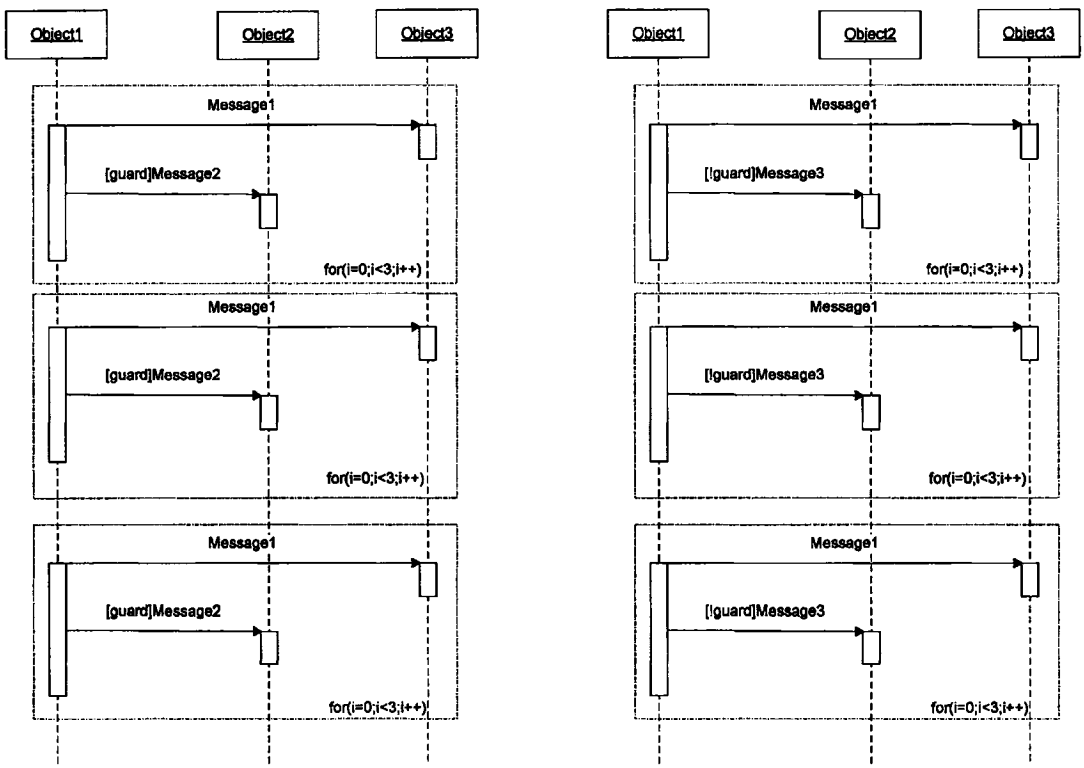


Figure 1 – Sequence Diagram Example



a) b)

Figure 2 – Two Possible Scenarios for Sequence in Figure 1

3.1 The Scenario Diagram Metamodel

We have adapted the UML metamodel [25], that is, the class diagram that describes the structure of scenario diagrams, to our needs, so as to ease transformation and the generation of sequence diagrams from traces.

The class diagram of the scenario diagram metamodel is shown in Figure 3. It has an abstract class `Message`, which has five derived classes: `Operation`, `Signal`, `ThreadStart`, `Create` and `Destroy`. `Message` has associations to the classes `Argument`, `Classifier`, `Repetition`, `Condition` and to itself. These are described in the sections below.

Note that there is no information related to distribution. Inter-node communication should be abstracted at this level, and so is displayed as any other message. The only indication shown will be the `nodeID` attribute of class `Class`. The value of this attribute will be different for each node, so if a message starts and ends from two objects or classes of different `nodeIDs`, this is a remote message.

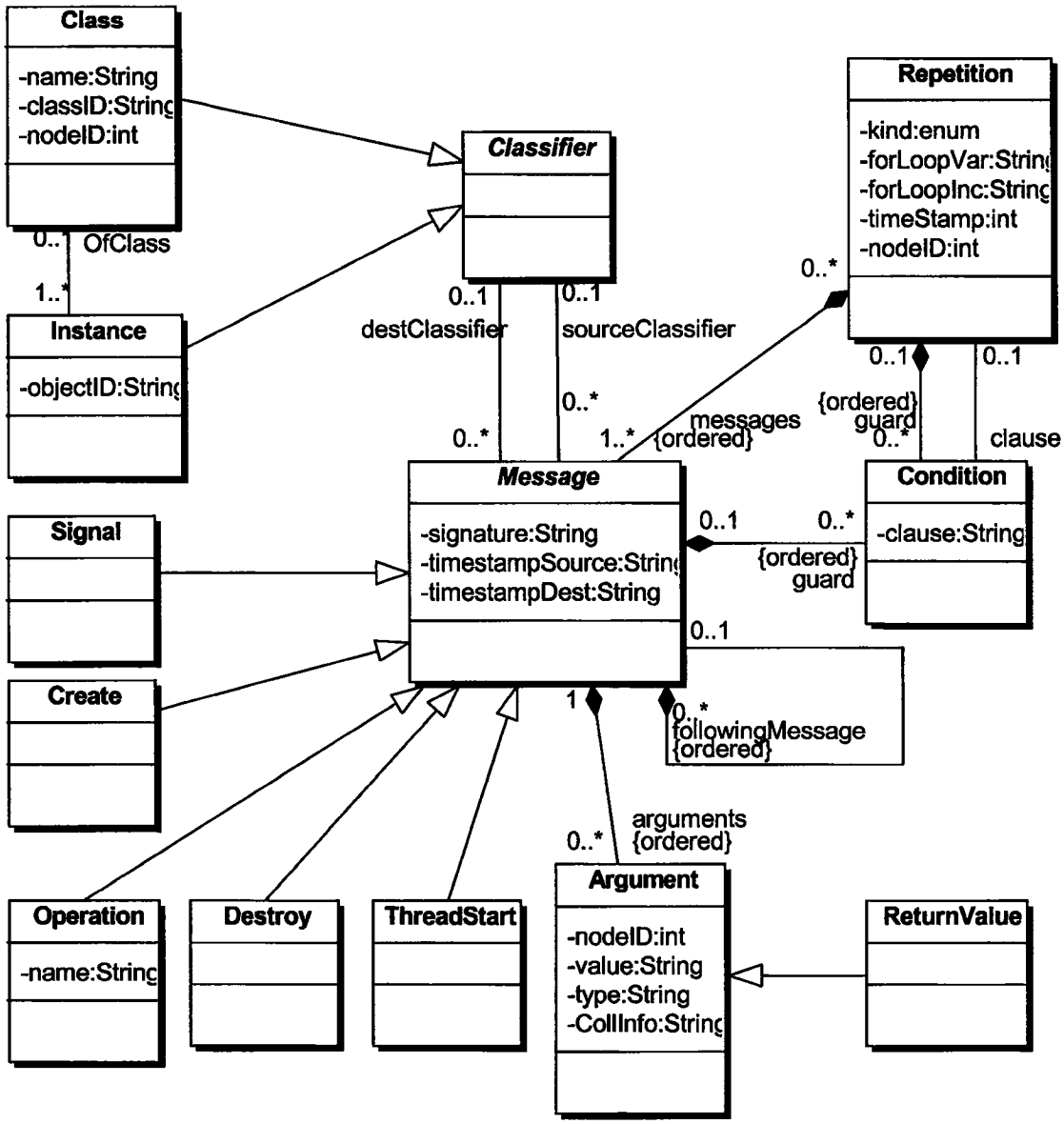


Figure 3 – Class Diagram of Scenario Metamodel

3.2 Messages

The class `Message` has the attributes `signature`, `timestampSource` and `timestampDest`. Though `signature` is self-explanatory, the timestamps merit more explanation, which is addressed in section 3.3. The following sections describe each subclass of `Message`.

3.2.1 Operation

The most basic interaction between objects shown in a sequence diagram is the method invocation. In a sequence diagram, this is shown as a filled arrow from one object's lifeline to another's. This is represented by the class `Operation` in our metamodel. It has the attribute `Name`.

3.2.2 Create and Destroy

Sequence diagrams have a different way of indicating the creation and destruction of an object, as opposed to an operation. At the creation of an object, the message points to the class role instead of the object's lifeline and is labeled with the stereotype <<create>> [4]. At the destruction of an object, the message points to a large "X", where the object lifeline ends. This message is labeled with the stereotype <<destroy>> [4]. The `Messages` `Create` and `Destroy` correspond to the creation and destruction of an object, respectively.

3.2.3 Signals

Signals are defined in UML as a named event that can be raised [11]. They are objects that describe an event that has occurred in the system. Often, these types of objects are described in a signal hierarchy in the system documentation. Signal objects are often used in sequence diagrams when two execution threads communicate through a common data structure. For example, in a consumer/producer relationship, the producer places a signal in a queue. Some time later, a consumer takes this object from the queue. This is shown in a sequence diagram as a signal (shown as a non-filled arrow [40] or as a half arrow in

sequence diagrams) from the producer to the consumer. It designates that the producer sent a message to the consumer without being blocked (see Figure 4). In our metamodel this is represented by the class `Signal`.

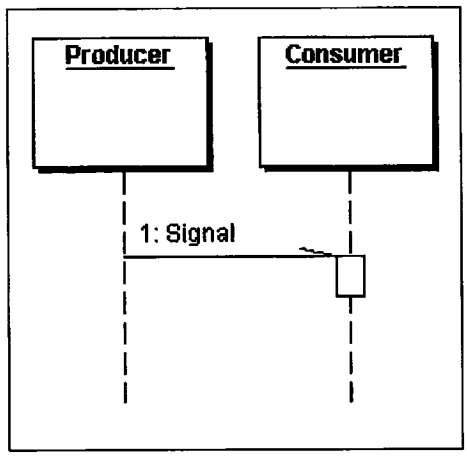


Figure 4 – Sequence Diagram Signal

3.2.4 ThreadStart

In a program with just one thread of execution, messages passed between objects are always synchronous. That is, the source of the message has to wait until the execution control is returned. For example, if method `m1()` calls a method `m2()`, `m1()` can continue to execute only once the method `m2()` returns. If there are multiple threads of execution, the messages can be synchronous or asynchronous. Certain constructs allow messages to be sent without blocking the sender. That is, if object 'A' triggers a certain asynchronous execution method on object 'B', 'A' is able to proceed with its execution immediately, as is the method in 'B'. This is an asynchronous message.

In many programming languages (ex. C++, Java), one asynchronous message available is the mechanism that starts a new thread of execution, here represented by the class `ThreadStart`. `ThreadStart` and `Signal` are the only classes that represent asynchronous messages. We do not include low level signals such as `notify()` as they are often specific to programming languages. Our scenario diagram metamodel abstracts out such low level details in either `ThreadStart` or `Signal` objects.

3.3 Following Messages

`Messages` can trigger other `Messages`. This is represented by the self-association `followingMessages` of `Message`. In a sequence diagram, a message points to an object's activation. The following messages are messages that start from this activation and go to another object. Each of these may or may not have following messages of their own. This is illustrated in Figure 5. The message numbered "1" has messages 1.1, 1.2 and 1.3 as its following messages.

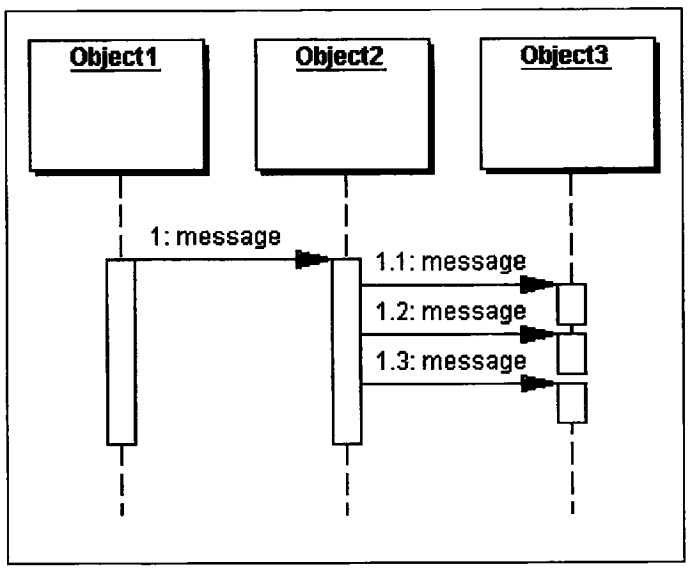


Figure 5 – Following Messages Example

Though the Messages associated with a Message are ordered, the order of the interactions between different threads of execution must be known. For example, when two threads are shown in the same sequence diagram, the position of a message relative to another message from the other thread is needed. The attributes timestampSource and timestampDest address this issue. These timestamps refer to the sending and receiving of the message, respectively. This allows us to know for example, when a method is executed, relative to another thread's Messages.

3.4 Arguments and Classifiers

Messages can have arguments (class Argument). The last argument can be the object returned from the message (subclass ReturnValue). Arguments have four attributes that are used in various ways to describe different kinds of objects. For primitives and types from the programming platform other than collections (e.g. Java library types), the type attribute contains the class name (or primitive name) and value contains the value returned by calling toString() on the object. In the case of an instance of a user defined class, or a Java collection instance, type, value and nodeID are used to uniquely identify the instance in the instrumented distributed system. The attribute value captures a unique identifier of the instance (for a given class), the type is the object's class name and nodeID uniquely represent nodes in the network, as further described in section 6.1.4.1. For collections, an extra attribute (collInfo) is provided to add information about the contents of the collection, either a list of the stored elements, the type of elements or their references. This is provided in order to view the value of the collection elements.

The Classifier class represents the source and destination of a message. They can be named objects (class Instance) or classes (class Class) in the cases where class scope methods are executed. The Instance class is an object associated with a type of Class, hence the association ofClass between the two classes. It is important to note here that each instance and each class must have a unique identifier. Class names, when they include the package name, must always be unique, so this is used as the identifier. Instances, on the other hand, are more complex. They may have many names because there may be several references to that object (i.e. aliasing). We assume here that each instance is given a unique number for each class (attribute objectID). For example, class Foo may have instances one, two and three, whereas class Bar has instances one and two. The implementation of the objectID depends on the SUS programming language and on the instrumentation strategy (see section Chapter 6). Once implemented, the objectID allows the unique identification of instances of a class. When paired with the className, it allows the unique identification of instances on a node. The combination of the objectID, the className and the nodeID, allows the unique identification of an instance in the distributed system. The classID shown in the metamodel is used purely for display, since the className can become quite long.

3.5 Conditions and Repetitions

A Message may have guard conditions, which must be true for the message to be sent. Though in UML sequence diagrams messages have only one guard, Message instances in our metamodel may have several to account for nested if statements, as discussed in 4.3. This composition has an {ordered} constraint since, when the Message has many

Conditions (nested if statements), the order in which they appear in the code is maintained. In the scenario diagram, these Conditions are combined in a conjunction that must be true for the Message to be sent.

A Message may also be part of a loop. This is represented by the class Repetition. A Repetition consists of a group of Messages that specifies which messages are repeated (composition between Repetition and Message). It has a kind of repetition (i.e. for, while loop) and a Condition representing the clause for which the loop stops repeating. A Repetition can also have guard conditions, in the same way as a Message. This is modeled by the guard association. Again, these Conditions are combined in an ordered conjunction that must be true for the Repetition to happen. The Repetition class also has a timeStamp attribute. These are explained further in Chapter 4. They are used to order nested loops. If two or more instances of Repetition have the same Messages, this signifies that these Messages are nested within more than one loop. The Repetition with the highest numbered timestamp will be the outermost box representing the iteration. The second outermost box will correspond to the second highest numbered timestamp, and so forth.

3.6 Example

The following is a simple example of a scenario diagram and the corresponding instance of the scenario metamodel. Figure 6 shows a scenario that starts with a message to Object1, which triggers the message message1 that has the argument 6, an int. Many messages follow this: a create message, message2, message3 (that has the condition A) and message4. Note that the while loop is only repeated once. Then message5 that has a

return value of `a`, a `String`, and a destroy message. The corresponding scenario metamodel instance is shown in Figure 7. The diagram may look complex, but this is due to the many associations, such as the ones between messages and classifiers and the `followingMessages` self-association.

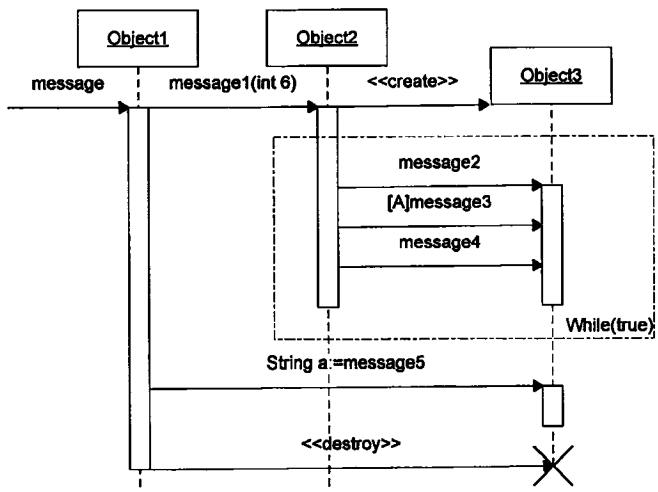


Figure 6 – Example Scenario Diagram

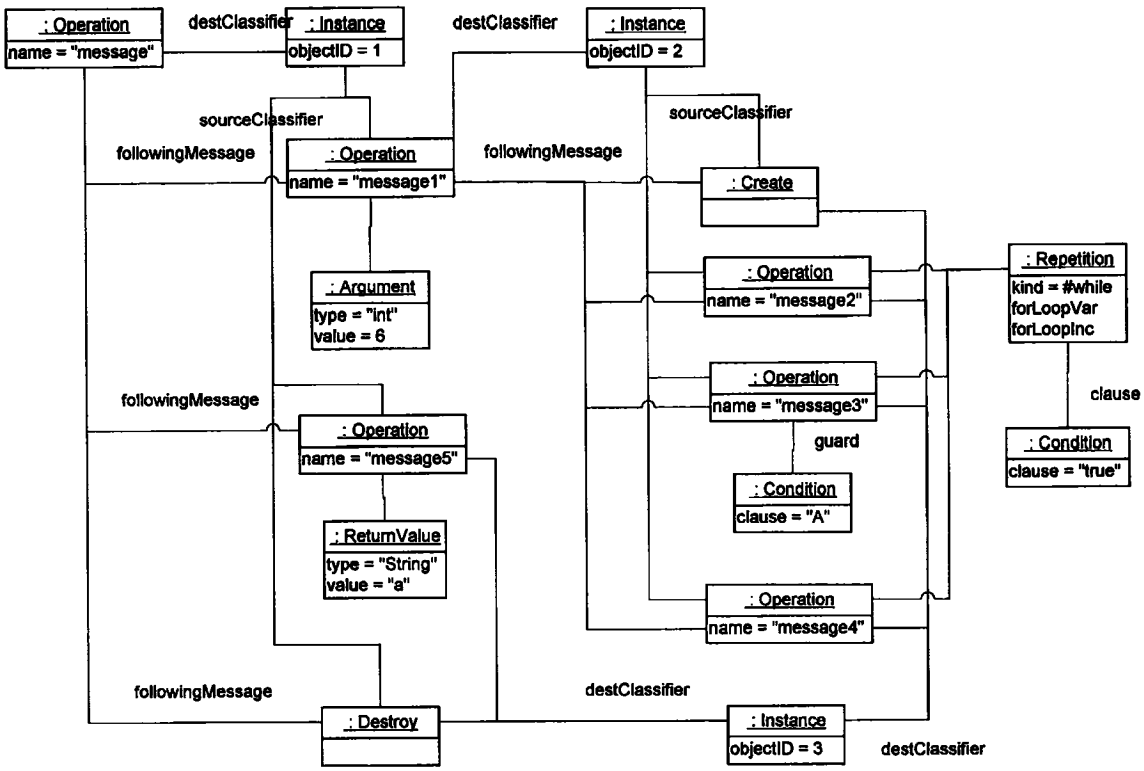


Figure 7 – Scenario Metamodel Instance Corresponding to Figure 6

CHAPTER 4 TRACE REPRESENTATION

As mentioned, we instrument the SUS by adding instrumentation statements in the source to retrieve the runtime information (see Chapter 6). These statements are automatically added and produce text lines (referred to as trace statements) in the trace file. The elements that we will need to instrument are: method entry and exit, conditions, loops, distributed information and concurrency related information. Each element will have one or more classes representing them in the trace metamodel. These elements are discussed in the next sections. We will see that the traces produced from the instrumentation statements are not in the form of messages needed for the scenario diagram. For this reason this chapter presents a metamodel for the trace and the next chapter presents the transformation rules to navigate between the two metamodels.

4.1 The Trace Metamodel

The class diagram of the trace metamodel is shown in Figure 8. The central class, `ExecutionStatement`, represents a pair of trace statements in the trace file. We say pairs here because as we will see in section 6.4.2, in most cases, an instance of `ExecutionStatement` has a defined beginning and end. There may be other statements in between; for instance, a method execution has a start and an end and may involve the execution of if statements. This relationship is modeled by the ordered self-association `nestedStatement-nestingStatement` of `ExecutionStatement`. `ExecutionStatement` is abstract and has three subclasses: `IfStatement`, `Repetition` and `MethodExecution`. These are discussed in the next sections.

Each trace statement, and so class `ExecutionStatement`, has a statement, a timestamp, a thread identifier and a node identifier. The statement attribute is the statement as it appears in the code. The timestamp (based on each node's local time) indicates when the trace statement occurred. It is used to order repetitions and signals and also acts as a unique identifier for each trace statement. See section 7.1 for details on how the timestamp is generated and section 4.5 for a discussion on why global timestamps are not required. The motivation behind the thread and node identifiers is discussed in sections 4.4 and 4.5 respectively.

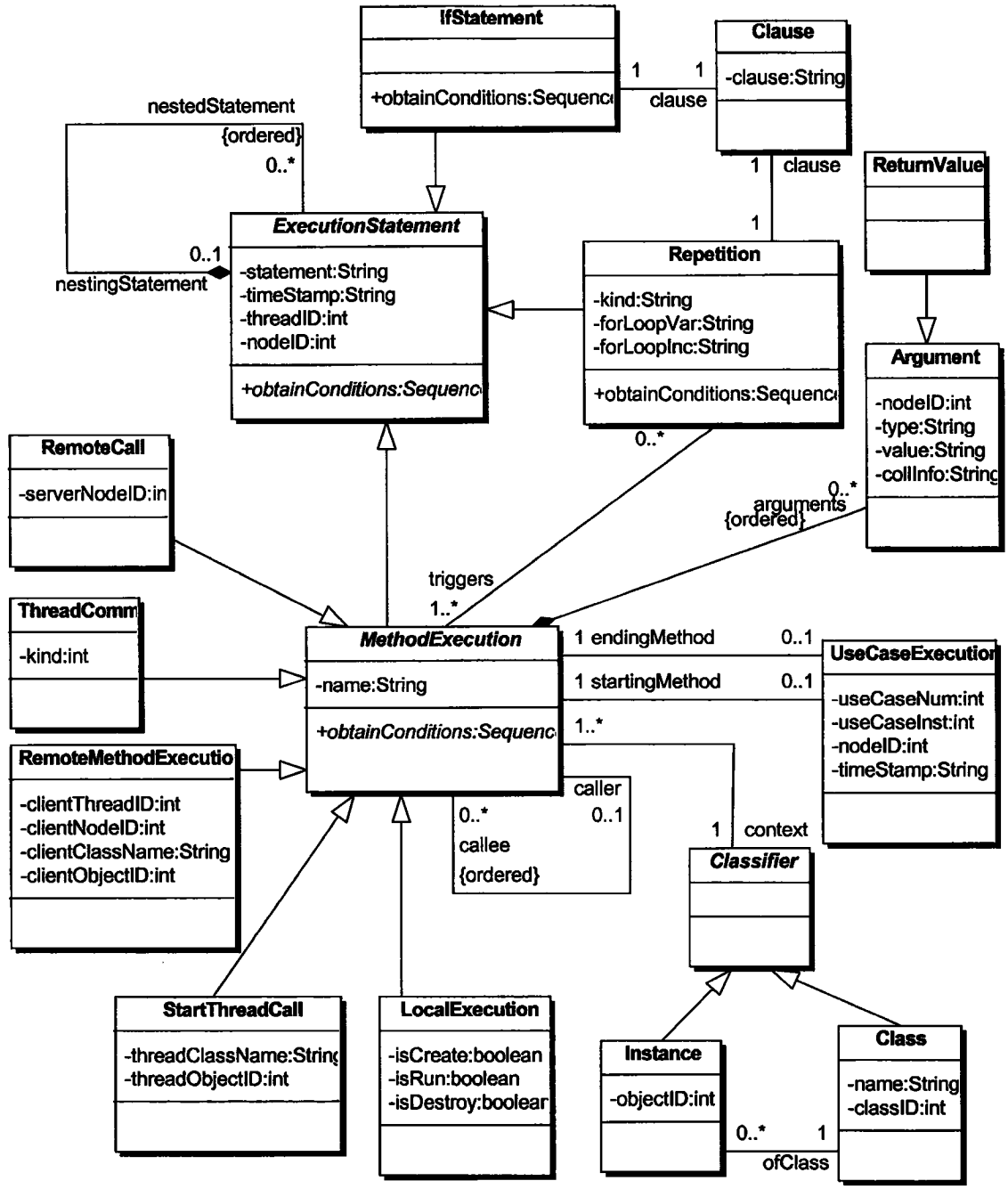


Figure 8 – Class Diagram of the Trace Metamodel

4.2 Method Entry and Exit

As stated previously, the most basic interaction between objects is the method invocation. Therefore, the beginning and end of each method must produce a trace statement to report these events. The statement must report on the signature, the object or class the method is executing on, the arguments and the return value, as well as the information all trace statements must report on (statement, thread identifier, node identifier and timestamp). In the trace metamodel, the execution of a method is represented by the abstract class `MethodExecution`.

`MethodExecution` has a composition association with the class `Argument`. This class and the `ReturnValue` subclass are identical to the classes of the same names in the scenario metamodel. They perform the same function: to represent the arguments and return value of the method. For each argument, the values of the attributes (`nodeID`, `type`, `value` and `collInfo`) must be within the trace statement produced by the method.

`MethodExecution` also has an association named `context` to the class `Classifier`. This class is very similar to the class of the same name in the scenario metamodel. The context of the method can be a named object (class `Instance`) or class (class `Class`) in the case where a class scope method is executed. However, the class `Class` in the trace metamodel does not have a `nodeID` attribute because it is already an attribute of the `ExecutionStatement`. Unlike the scenario metamodel, `MethodExecution` only has access to the object (or class) that the method is executing on. When transforming a trace metamodel instance to a scenario metamodel instance, this context will correspond to the

destination of the message. The source will be the context of the method's caller. This brings us to the explanation of the `caller-callee` self-association of `MethodExecution`.

Though the `caller-callee` association is not read directly from the trace file, it is in the trace metamodel in order to simplify navigation, such as when seeking a method's caller. This association is redundant since the information to create the link is contained in `ExecutionStatement`'s `nestedStatement-nestingStatement` association, therefore the link can be set offline (i.e. once the rest of the metamodel is instantiated). As the name suggests, the association links each caller of a method to all of its callees. A method has a link to all the methods it calls directly. It ignores any conditions or loops (see section 4.3) that may surround a method call.

Figure 9 shows an example of code and the corresponding trace instance. We assume here that method `mA()` of class `A` was called, that conditions `c1` and `c2` are true and that there is only one execution of the while loop. `mA()` has one nested statement, `if(c1)`, that in turn has one nested statement `while(c2)`, that in turn has the nested statement `theB.mB()`. To have a direct association with its caller, we have the association between `mA()` and `mB()`.

In order to build this link, one must examine the nested statements of an instance of `MethodExecution`. Each of those that are instances of `MethodExecution` as well must have a caller-callee link. Those that are not must recursively call all their nested statements for instances of `MethodExecution` and so on. In our example, we first search the `nestedStatements` of the `MethodExecution` representing the method `mA()`. There is only one, an `IfStatement` instance. Since this is not a `MethodExecution`, we search its

nestedStatements, which there also only one, a Repetition instance. When we search its nestedStatements, we find a MethodExecution instance representing the method mB(), so we create a caller-callee link between mA() and mB().

The subclass LocalExecution of MethodExecution represents a regular method execution: method calls inside the same virtual machine and the same thread. It has the attributes isRun, isCreate and isDestroy of type Boolean. These, of course, are flags that indicate whether the method is the run method of a thread, a constructor or a destructor, respectively. Other subclasses of MethodExecution are discussed in the sections 4.4 and 4.5.

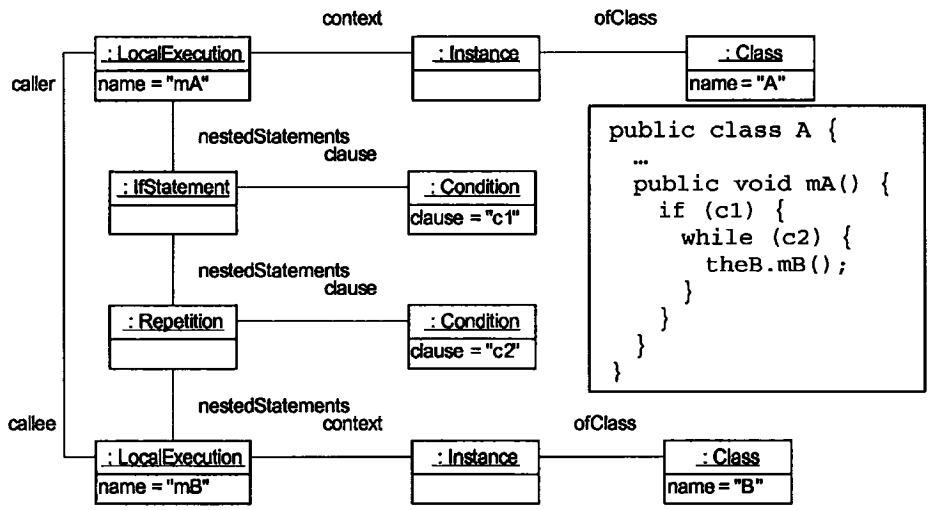


Figure 9 – Instance Example of the Trace Metamodel

4.3 Conditions and Repetitions

Control flow information must also be present in the trace. Since conditions and loops have a beginning and an end, as well as other statements in between, the classes

`IfStatement` and `Repetition` are subclasses of `ExecutionStatement`. Both these classes have an association to the class `Condition`, which represents the clause.

The class `Repetition` also has the attributes `kind`, `forLoopVar` and `forLoopInc`. The kind is the type of loop: either `while`, `dowhile` or `for`. In the case of a "for" loop, more information is need other than the clause, such as the variable used in the loop and the increment to the variable between each repetition, which are stored in the corresponding attributes. This information must be contained in the trace. Loops are sometimes ended prematurely by a break or a continue statement. These are valid trace statements, but are not represented in the trace metamodel since these are the equivalent to the end of a loop.

`Repetition` has a `triggers` association to the class `MethodExecution`. It is very similar to the `caller-callee` association in that it is redundant information and it is not found in the trace directly, but provides a practical way of navigating the metamodel. The link represents each method execution that is called within a repetition. Again, this association only goes one level deep; for example, a method `m()` that is called within a while loop is linked to the corresponding `Repetition` instance, but any methods called within `m()` are not. This association is built through a recursive traversal of the nested statements: for each nested statement of the `Repetition`, the instances that are `MethodExecutions` are linked. The ones that are not, their nested statements are searched for `MethodExecutions` and linked, and so on. The result is a link between a repetition and all the methods triggered within.

A condition is another element that is in the trace. If a message has a condition, the message can be sent only when this condition has been fulfilled. This information is

important to the scenario diagram, therefore the trace may contain conditions such as if, else if, else, end if and switch statements. These are all modeled by the class `IfStatement`, the third and last subclass of `ExecutionStatement`. In the trace metamodel, "if" statements are similar to repetitions: they have nested statements and a clause. However, this is not true in the scenario metamodel, where the class `Condition` is part of `Message` and of `Repetition` through composition associations (see section 3.5). This is why there is not an equivalent of `Repetition`'s `triggers` association to `MethodExecution`. Instead, it would be useful to have a mechanism to query an `ExecutionStatement` to obtain all of the conditions its execution depends on. This is especially helpful when there are several nested "if" statements, therefore the complete condition that a statement is subject to is not directly available. This is accomplished by the operation `obtainConditions()` in class `ExecutionStatement`. The post-condition of `obtainConditions()` in classes `MethodExecution`, `Repetition` and `IfStatement` can be found in Table 3.

Table 3 – Post-conditions for `obtainConditions()` in classes `MethodExecution`, `Repetition` and `IfStatement`

```
context MethodExecution::obtainConditions():Sequence(Condition)
post: if (nestingStatement.oclIsTypeOf(IfStatement) then
            result = nestingStatement.obtainConditions()
      else
            result = null
```

```
context Repetition::obtainConditions():Sequence(Condition)
post: if (nestingStatement.oclIsTypeOf(IfStatement) then
            result = nestingStatement.obtainConditions()
      else
            result = null
```

```
context IfStatement::obtainConditions():Sequence(Condition)
post: if (nestingStatement.oclIsTypeOf(IfStatement) then
            result = nestingStatement.obtainConditions().
                  append(self.Clause)
      else
            result = self.Clause
```

4.4 Concurrency

One of our goals is to allow the SUS to be concurrent. This has a significant impact on the instrumentation and the trace. First, the trace statements produced by one thread will be interleaved with other threads' trace statements. Second, when one thread starts another, we must be able to match the caller and the new thread (despite the fact that they are in separate threads of execution) because this is a type of message. Third, threads may communicate with each other through signals. These issues are discussed below.

4.4.1 Multi-Threaded Environments

In a concurrent system, many threads of execution may be running simultaneously. Whether in a multi-processor environment where the threads are truly running at the same time, or whether the threads are sharing the same processor and so are scheduled, the impact is the same: trace statements produced by instrumented events from one thread will be interwoven with the trace statements of other threads. It is important to distinguish trace statements of each thread from the others because the separate sequence of statements must be maintained for each thread.

Our solution is to assign a unique identifier for each thread of execution. When reading the trace, statements from different threads can be easily distinguished from all other threads and thus the order of the statements in each thread is preserved. This field must be in each trace statement and will be stored in the trace metamodel through the `threadID` attribute of `ExecutionStatement`. The implementation of this identifier will depend on the programming language of the SUS and of the instrumentation strategy (see section 6.2).

4.4.2 Starting Threads

As discussed in section 3.2.4, many programming languages have a mechanism to start new threads of execution. In Java, one must call the `start()` method of a thread object. The virtual machine then triggers the object's `run()` method in a different thread of execution. Because this is a type of message, the call to the `start()` method, not just the execution of the `run()` method, must produce trace statements.

In the trace metamodel, the call to a `start()` method is represented by the class `StartThreadCall`, a subclass of `MethodExecution`. The execution of a `run()` method is a `LocalExecution` with the flag `isRun` set to true.

As discussed above, the SUS may be concurrent and trace statements may be interwoven. If there are many `start()` calls and `run()` executions, we must have a mechanism to identify which `start()` method corresponds to which `run()` method (and so which `StartThreadCall` instance corresponds to which `LocalExecution` instance). These trace statements will have different `threadIDs` because they belong to different threads of execution, therefore we cannot use this information to do the matching. However, all that is needed to do the match is include the thread object's `objectID` and class name in the trace statement when instrumenting the call to the `start()` method. Because a thread's `run()` method can only be called once, this is a unique match. The `StartThreadCall` instance stores the thread's information in the attributes `threadObjectID` and `threadClassName`. These must match the `LocalExecution`'s `context` attributes.

4.4.3 Thread Communication

As described in section 3.2.3, threads may communicate with each other though signals. This may be very difficult to instrument: what is considered a signal? Something as simple as changing an attribute's value could be considered as thread communication. It may be too unpredictable to detect when communication between threads occurs because not all thread events are asynchronous communication with other threads. For this reason, we propose the following approach: we will detect when signals are placed in a specific data structure. In other words, the object being placed in the data structure will be an object from the signal hierarchy, which will be provided by the system documentation. The data structure must be either an object in the SUS or an object from the Java library that the user has identified in advance as a possible structure for the signals. The data structure methods must also be identified in advance to specify if it is a "read" or "write" to the structure. In the case of the read, the return object must be of a type from the signal library, and in the case of the write, the first argument must be this same object. Calls to those methods will be instrumented to produce trace statements that will be represented by the class `ThreadComm` in the trace metamodel. Matching read and write `ThreadComm` instances will result in a `Signal` instance in the scenario metamodel. This is illustrated in Figure 10a and Figure 10b for a Producer-Consumer relationship. The data structure is a `Queue`, the write method is `put()` and the read method is `remove()`.

This approach may seem like a significant limitation, but in fact it is considered good practice that specific design patterns be used to implement real-time systems [9, 12] such as using a FIFO queue, and that data exchanged asynchronously be modeled as instances of signal classes belonging to an inheritance hierarchy [4, 11].

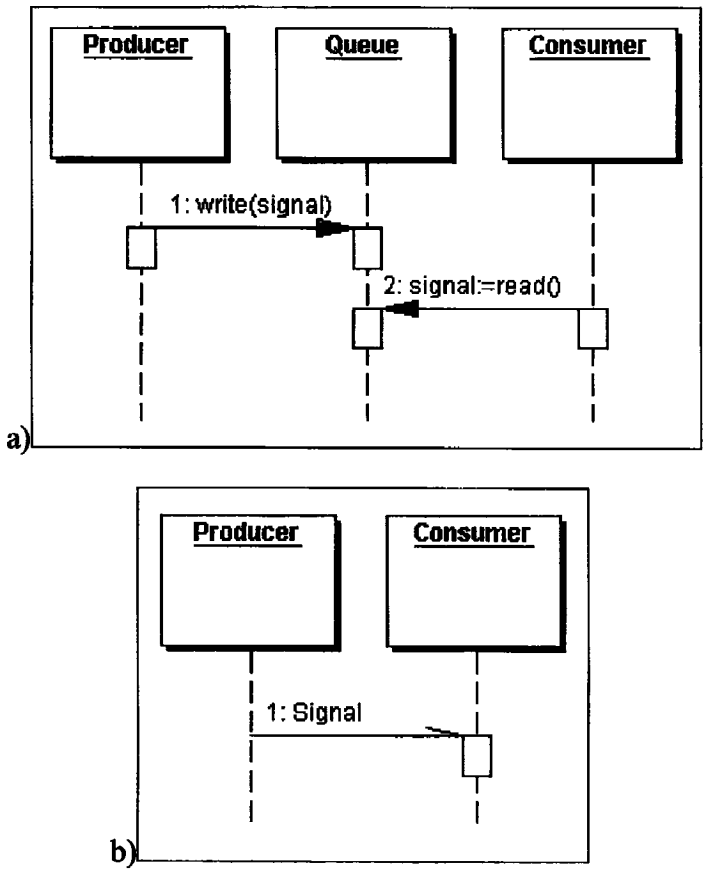


Figure 10 – a) Generated Trace Sequence Diagram b) Desired Diagram

4.5 Distributed Information

Another of our goals is to allow the SUS to be distributed. Many nodes in a network can communicate with each other through remote method calls. Method entry and exit instrumentation statements as described in section 4.2 are not sufficient in the case of distributed systems since it will not be known if there has been a call to a remote object. That is to say, in the trace produced by instrumentation statements for one component, a call to a remote method might have occurred, but the method entry and exit

instrumentation statements will appear in the trace of the remote component. This will make it difficult to reconstruct the order in which the call took place since there are two separate traces. With only access to the traces, and not to the code, the trace created by the remote call may be inserted in many possible points in the caller's trace.

To solve this problem, there must be some indication in the caller's code that a remote call is being made. Note that not all method calls need to be instrumented: only remote method calls must be identified to add instrumentation statements around them. Remote method calls are represented in the trace metamodel by the class `RemoteCall`. However, instrumenting the calls alone is sufficient only in the situation where there are only two nodes: one server and one client. If there is one server but several clients, we have the scenario where the server's method executions could match any of the clients' remote calls. The reverse is also an issue. If there are many servers and one (or more) client(s), each of the servers' method executions could match any of the client's remote calls. Recall that a client may have many threads of execution. This adds yet another problem: even with one client and one server, several threads may perform remote method calls. Again, matching the client remote calls to the correct server method execution is a concern.

We have adopted the following approach to solve this problem: clients and servers require information about each other. When a client sends a request to a server (a remote call), it must know exactly which server it is communicating with. Equally, the server must know exactly which client it is serving. For that reason, each node must be uniquely identified to distinguish them from each other. This is the purpose of the `nodeID`, which is assigned to every node. This is an attribute of `ExecutionStatement`.

The class `RemoteCall` has the attribute `ServerNodeID` that, obviously, stores the `nodeID` of the call's target. On the server side, we introduce a new class: `RemoteMethodExecution`. This class wraps around the server method (around the `LocalExecution` corresponding to the execution of the server method). It has the attributes `ClientNodeID` and `ClientThreadID`. They store the `NodeID` and `ThreadID` of the client it is serving. As discussed below, this is enough to match `RemoteCall` instances to `RemoteMethodExecution` instances. We also include information about the client's objectID and class name. Though these are not required, they are added to facilitate searches through the metamodel. The `RemoteMethodExecution` and the `LocalExecution` are linked together by the `caller-callee` association. With all this added information, the matching pairs of remote method calls and remote method executions are unique.

We may encounter a situation where a node (and one thread) makes many remote calls to the same server. Recall that RMI calls are synchronous, that is when a thread at the client side performs a call to a remote method, it blocks until the remote execution terminates. As a consequence, the order of calls to remote methods (at the client side) corresponds to the order of remote method executions (at the server side), even though the requests are serviced by different threads on the server side. The calls will be ordered by the client's local timestamp. The server's method executions will also be ordered by its local timestamp. Therefore, the first call corresponds with the first server method execution and so on. Local timestamps, the additional information about clients and servers as well as the fact that the remote communication is synchronous all combine to match remote calls to the correct remote method execution using only the information discussed above.

This methodology applies only in the ideal case where remote method calls are successful and no messages are lost. A solution to this would be to send the caller's timestamp along with its thread and node identifiers. However, this is not implemented in the present work.

4.6 Use Cases

During a scenario, many use cases may be running at the same time. It might be interesting to know when a use case starts and stops. This is represented by the class `UseCaseExecution` in the trace metamodel. It has two associations to `MethodExecution`: `startingMethod` and `endingMethod`. This information could be used at a later point when abstracting scenarios because there should be a generalized sequence diagram for each use case.

Use case numbers to identify each use case uniquely must be created by the system. The proposed approach is to interact with the user before instrumentation to identify which methods are considered to start a use case. Those methods can create a new `UseCaseExecution` instance every time it is executed. This proposition holds the assumption that these methods will not be called in the middle of a use case execution. This is a reasonable assumption since it is good programming practice to separate objects into entity, boundary and control objects [7].

Boundary objects should contain a method that is a good candidate for this proposition. Despite this restriction, this approach has the advantage of having two parts to identify a use case: one number to identify the use-case that is executing (attribute `useCaseNum`) and another to number each execution of the same use-case (attribute `useCaseInst`). For

example, one method could be designated 'use-case #3', and the first time it is called it will create a `UseCaseExecution` instance that contains a `useCaseNum` equal to three and `useCaseInst` equal to one. The second time it is executed, the `useCaseNum` will equal three and the `useCaseInst` will equal two, and so on. This can be done offline because the appropriate methods can be searched in the trace metamodel instance.

CHAPTER 5 CONSISTENCY RULES

We have derived five consistency rules, expressed in 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, though they provide a specification and insights into how implementing 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. Note that Appendix A shows more complicated examples of scenario metamodel instances obtained from trace metamodel instances. Three consistency rules have been defined to match `Message` child classes from instances of `MethodExecution` child classes (Section 5.1), one consistency rule has been defined to identify links between `Message` instances, i.e., association `followingMessage` (Section 5.2), and one consistency rule has been defined to identify repetitions of `Message` instances (Section 5.3).

5.1 Identifying instances of `Message`

The first consistency rule describes the mapping between instances of `Trace::MethodExecution` child classes and instances of `Scenario::Message` child classes (Figure 12). The most common situation occurs when two instances of `LocalExecution` are related by a `caller-callee` link (see Figure 8), as this corresponds to an instance of `Message` child class `Operation`. The rule also handles all the other child classes of `Trace::MethodExecution` and `Scenario::Message`.

The first six lines of the consistency rule in Figure 12 state that whenever two instances of `Trace::MethodExecution`, `me1` and `me2`, satisfy either of four conditions, there exists a corresponding instance of `Scenario::Message`. The four conditions are modeled as Boolean query operations to simplify and improve the readability of our OCL expressions: `remoteCall(me1, me2)`, `localCall(me1, me2)`, `startThread(me1, me2)` or `threadComm(me1, me2)`. All four operations have two parameters of type `MethodExecution`, and their pre and post conditions can be found in Figure 13. In our tool prototype, they are implemented in utility class `CheckMapping`, which does not appear in our metamodel, but which will be referred to in OCL expressions making use of these operations.

Operation `localCall(me1, me2)` then returns true 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. Operation `remoteCall(me1,me2)` returns true when `me1` and `me2` are instances of `RemoteCall` and `RemoteMethodExecution`, respectively. Additionally, the attributes of `me1` and `me2` must match (values of `threadID`, `nodeID`, `serverNodeID` and `context`), that is, `me1` is a call on the RMI client side of a remote method and corresponds on the server side to execution `me2`. This situation is further illustrated by a typical example trace metamodel instance in Figure 11(a). Note that the instance of the trace metamodel created for the client (one trace file) is not linked in any way to the instance of the trace metamodel created for the server (another trace file). The purpose of the transformation of the trace metamodel instance into a scenario metamodel instance is to abstract this situation by transforming instances of `RemoteCall`

and `RemoteMethodExecution` into a `Message` instance between the two related instances of `LocalExecution`.

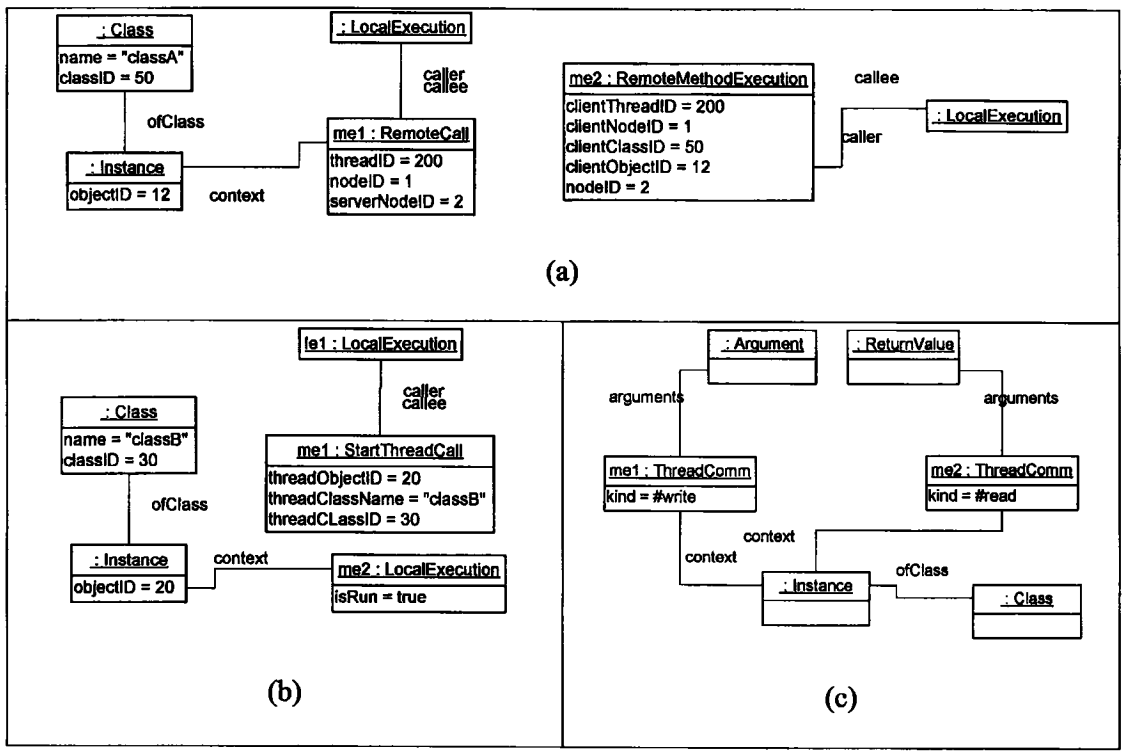


Figure 11 – Examples of the first consistency rule

Operation `startThread(me1,me2)` returns true when `me1` is a `StartThreadCall` instance and `me2` is an instance of `LocalExecution` whose attribute `isRun` is true (i.e., it is a `run()` execution). Furthermore, the attributes of `me1` and `me2` must match: attributes `threadClassName`, `threadClassID` and `threadObjectID` of `me1` must match `me2` attributes `className`, `classID`, and `objectID`, indicating that the call to start corresponding to `me1` actually triggers the execution of method `run()` corresponding to `me2`. Figure 11(b) illustrates the situation on an example. Note again that, though only one trace file is involved here as the two threads execute on the same node in the network, the instance of the trace metamodel contains two separate parts.

Operation `threadComm(me1,me2)` returns true when `me1` writes a signal in a data structure and that signal is read by `me2`. As described and justified in Section 4.4.3, we assume asynchronous thread communications are performed by means of data structures that contain instances of `Signal` classes. In other words, these data structures are observed at runtime and operations that write or read `Signal` instances to and from them are caught, resulting in instances of class `ThreadComm`, of `kind write` or `read`, respectively. This is the reason why `CheckMapping::threadComm(MethodExecution, MethodExecution)` does not verify that the parameter of `me1`, returned by `me2`, is a `Signal`: since we have `threadComm` instances, we know it is the case (this is ensured during instrumentation). In order to abstract an asynchronous thread communication from `MethodExecutions` `me1` and `me2` (thus executing in two different threads), `me1` and `me2` must then be `"write"` and `"read"` `ThreadComm` instances with the same `context` (i.e., the data structure involved, as modeled by an association in the trace metamodel). Furthermore, `me1`'s unique signal argument, i.e., the signal sent by the thread executing `me1`, must be the signal returned by `me2`. Again, a typical situation is illustrated in Figure 11(c). Note that the argument and the returned value are two different instances. However, the OCL equality between instances is a logical equality, i.e., the `objectID` attribute of the two instances must have identical values to correspond to the same signal. Note that several `ThreadComm` instances of kind `read` can be associated with a single `ThreadComm` instance of kind `write`, as long as they all manipulate the same `Signal` instance in the same data structure. This corresponds to an asynchronous message sent to more than one thread.

The rest of the consistency rule in Figure 12 ensures that the attributes of `me1` and `me2`, as identified by either of the four query operations, and the matching `Message` instance `mes` are consistent. First, `timestamps` of `me1`, `me2` and `mes` are checked: `mes.timestampDest` equals to `me2.timestamp` and `mes.timestampSource` equals to `me1.timestamp`, except in the case of a local call, where both `timestamps` of `mes` are equal to `me2.timestamp`. Next, `contexts` of `me1` and `me2` correspond to source and target classifiers of `mes`, respectively, exept in the case where `me1` and `me2` are instances of `ThreadComm`. This connection is checked using operation `mapContextClassifier()` which postcondition is provided in Figure 14. The rule also ensures that the arguments of `mes` match the ones of `me2` (using operation `mapExecutionMessageArgs()` whose post condition is provided in Figure 14). The mapping between `me1` or `me2` conditions (using operation `obtainConditions()`) and `mes` guard condition is then verified. In case `localCall(me1,me2)` is true, `mes` guard is `me2.obtainConditions()`. However, in all the other three situations, the guard is `me1.obtainCondition()`, since the conditions that lead to the sending of `mes` are linked with `me1` in the trace metamodel instance. For instance, if an asynchronous message is sent between two threads, an instance of `ThreadComm` (`kind=#write`) appears in the trace metamodel instance and is associated with a `Condition` instance. It is this condition that decides of the sending of the message. Last, the signature, name and actual type (child class of `Message`) of the message are checked. For instance, when `localCall(me1,me2)` is true, if `me2` is a constructor or a destructor, so is `mes` (instance of `Create` or `Destroy`) otherwise `mes` is an instance of `Operation`.

Last, note that 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 any of the four query operations we used. 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 non instrumented subsystems (e.g., a GUI subsystem). These `LocalExecution` instances are nevertheless mapped to `Scenario::Operation` instances, though the message does not have a source classifier. Similarly, a trace metamodel instance may not contain any `LocalExecution` instance with attribute `isRun` equal to true corresponding to a given `StartThreadCall` instance. Another similar case is when the sender or receiver of a signal is detected, but not both. The corresponding mappings for the above special cases are the purpose of consistency rules in Figure 15 and Figure 16. Since they are not very different from Figure 12, they are not further described here.

```
Trace::MethodExecution.allInstances->forAll( me1: Trace::MethodExecution,
                                             me2: Trace::MethodExecution |
  CheckMapping.localCall(me1,me2)  or  CheckMapping.remoteCall(me1,me2)
  or  CheckMapping.startThread(me1, me2)  or  CheckMapping.threadComm(me1, me2)
  implies
  Scenario::Message.allInstances->exists(mes: Scenario::Message |
    //timestamps
    if CheckMapping.localCall(me1, me2) then (
      mes.timestampSource = me2.timestamp
      and mes.timestampDest = me2.timestamp
    ) else (
      mes.timestampSource = me1.timestamp
      and  mes.timestampDest = me2.timestamp
    ) endif
    and
    if CheckMapping.threadComm(me1, me2) then (
      //the context of me1's caller is the source of message mes
      CheckMapping.mapContextClassifier(me1.caller.context, mes.sourceClassifier)
      and
      //the context of me2's caller is the target of message mes
      CheckMapping.mapContextClassifier(me2.caller.context, mes.destClassifier)
    ) else (
      //the context of me1 is the source of message mes
      and  CheckMapping.mapContextClassifier(me1.context, mes.sourceClassifier)
      //the context of me2 is the target of message mes
      and  CheckMapping.mapContextClassifier(me2.context, mes.destClassifier)
    )
    //compare arguments (matching entire sequences)
    and  CheckMapping.mapExecutionMessageArgs(me2, mes)
    and //mapping the message guard to me1 or me2 conditions
    if CheckMapping.localCall(me1,me2) then (
      mes.guard.clause = me2.obtainConditions()
    ) else (
      mes.guard.clause = me1.obtainConditions()
    ) endif
    //mapping the exact message type, along with message name and signature
    and  CheckMapping.localCall(me1,me2) implies (
      mes.signature = me2.statement  and  mes.name = me2.name
      and  me2.isCreate = true implies mes.oclType = Scenario::Create
      and  me2.isDestroy = true implies mes.oclType = Scenario::Destroy
      and  not (me2.isCreate = true or me2.isDestroy = true) implies
             mes.oclType = Scenario::Operation
    )
    and  CheckMapping.remoteCall(me1,me2) implies (
      mes.signature = me2.statement  and  mes.name = me2.name
      and  mes.oclType = Scenario::Operation
    )
    and  CheckMapping.startThread(me1, me2) implies (
      mes.oclType = ThreadStart  and  mes.signature = me2.statement
    )
    and  CheckMapping.threadComm(me1, me2) implies (
      mes.oclType = Signal
    )
  ) // Scenario::Message.allInstances->exists
)
```

Figure 12 – Mapping `Trace::MethodExecution` instances to `Scenario::Message` instances

```
context CheckMapping::localCall( le1: MethodExecution,
                                 le2: MethodExecution ): Boolean
post: result = le1.oclType = LocalExecution  and  le2.oclType = LocalExecution
              and  le1.callee->includes(le2)
```

```
context CheckMapping::remoteCall( rc: MethodExecution,
                                  rme: MethodExecution ): Boolean
post:  result =
   rc.oclType = RemoteCall  and  rme.oclType = RemoteMethodExecution
   and  rc.serverNodeID = rme.nodeID  and  rc.threadID = rme.clientThreadID
   and  rc.nodeID = rme.clientNodeID
   and  if rc.context.oclType = Trace::Instance then (
     rc.context.objectID = rme.clientObjectID
     and  rc.context.ofClass.classID = rme.clientClassID
   } else (
     rc.context.classID = rme.clientClassID
   ) endif
```

```
context CheckMapping::startThread( stc: MethodExecution,
                                   le: MethodExecution ): Boolean
post:  result =
   stc.oclType = StartThreadCall  and  le.oclType = LocalExecution
   and  le.isRun = true  and  stc.threadClassName = le.context.ofClass.name
   and  stc.threadClassID = le.context.ofClass.classID
   and  stc.threadObjectID = le.context.objectID  and  stc.nodeID = le.nodeID
```

```
context CheckMapping::threadComm(  tc1: MethodExecution,
                                   tc2: MethodExecution ): Boolean
post:  result =
   tc1.oclType = ThreadComm  and  tc1.kind = #write
   and  tc2.oclType = ThreadComm  and  tc2.kind = #read
   and  tc1.arguments->at(1) = tc2.returnValue  and  tc1.context = tc2.context
```

Figure 13 – Postconditions for operations `localCall()`, `remoteCall()`, `startThread()` and `threadComm()` in class `CheckMapping`

```
context CheckMapping::mapContextClassifier( co : Trace::Classifier,
                                            cl : Scenario::Classifier): Boolean
post:  result =
   if (co.oclType =  Trace::Instance) then (
     cl.oclType = Scenario::Instance  and  cl.objectID = co.objectID
     and cl.ofClass.name = co.ofClass.name
     and cl.ofClass.classID = co.ofClass.classID
     and cl.ofClass.nodeID = co.ofClass.nodeID
   ) else (
     cl.oclType = Scenario::Class  and  cl.name = co.name
     and cl.classID = co.classID  and  cl.nodeID = co.nodeID
   ) endif
```

```
Context CheckMapping::mapExecutionMessageArgs(me : Trace::MethodExecution,
                                              m : Scenario::Message): Boolean
post:  result =
   m.arguments.nodeID = me.arguments.nodeID
   and m.arguments.value = me.arguments.value
   and m.arguments.type = me.arguments.type
   and m.arguments.collInfo = me.arguments.collInfo
   and Sequence{1..me.arguments->size}->forAll(index: Integer |
     if me.arguments->at(index).oclType = Trace::ReturnValue
     then m.arguments->at(index).oclType = Scenario::ReturnValue
     else m.arguments->at(index).oclType = Scenario::Argument endif
   )
```

Figure 14 – Post condition of operation `CheckMapping::mapContextClassifier()` and `CheckMapping::mapExecutionMessageArgs()`

```
Trace::MethodExecution.allInstances->forAll( me2: Trace::MethodExecution |
  MethodExecution.allInstances->select(me1: Trace::MethodExecution |
    CheckMapping.localCall(me1,me2) or CheckMapping.startThread(me1, me2)
    or CheckMapping.threadComm(me1, me2)
    )->Empty
  implies   //me2 does not have a caller => message without source
  Scenario::Message.allInstances->exists(mes: Scenario::Message |
    //timestamps
    mes.timestampSource = me2.timestamp
    and  mes.timestampDest = me2.timestamp
    //there is no source
    and  mes.sourceClassifier = null
    //the context of me2 is the target of message mes
    and  CheckMapping.mapContextClassifier(me2.context, mes.destClassifier)
    //compare arguments (matching entire sequences)
    and  CheckMapping.mapExecutionMessageArgs(me2, mes)
    //we do not have the caller, thus no access to the conditions/guard
    and  mes.guard = null
    //mapping the exact message type, along with message name and signature
    and  me2.oclType = LocalExecution and me2.isRun = false implies (
      mes.signature = me2.statement  and  mes.name = me2.name
      and  me2.isCreate = true implies mes.oclType = Scenario::Create
      and  me2.isDestroy = true implies mes.oclType = Scenario::Destroy
      and  not (me2.isCreate = true or me2.isDestroy = true) implies
             mes.oclType = Scenario::Operation
    )
    and  me2.oclType = LocalExecution and me2.isRun = true implies (
      mes.oclType = ThreadStart  and  mes.signature = me2.statement
      and mes.name = me2.name
    )
    and  me2.oclType = ThreadComm implies (
      mes.oclType = Signal  and  mes.name = me2.returnValue.name
    }
  ) // Scenario::Message.allInstances->exists
)
```

Figure 15 – Incomplete mapping of `Trace::MethodExecution` instances to `Scenario::Message` instances[5]

[5] Note that because of our instrumentation strategy, not having a `RemoteCall` instance (the caller) for a given `RemoteMethodExecution` is not possible. Similarly, not having a `RemoteMethodExecution` instance for a given `RemoteCall` instance is not possible.

```
Trace::MethodExecution.allInstances->forAll( me2: Trace::MethodExecution |
  MethodExecution.allInstances->select(me1: Trace::MethodExecution |
  CheckMapping.startThread(me2, me1)  or  CheckMapping.threadComm(me2, me1)
  )->isEmpty
  implies   //me2 does not have a callee => message without destination
  Scenario::Message.allInstances->exists(mes: Scenario::Message |
    //timestamps
    mes.timestampSource = me2.timestamp
    and  mes.timestampDest = me2.timestamp
    //the context of me2 is the source of message mes
    and  CheckMapping.mapContextClassifier(me2.context, mes.sourceClassifier)
    //there is no target
    and  mes.destClassifier = null
    //compare arguments (matching entire sequences)
    and  CheckMapping.mapExecutionMessageArgs(me2, mes)
    //mapping the message guard to me1 or me2 conditions
    mes.guard.clause = me1.obtainConditions()
    //mapping the exact message type, along with message name and signature
    and  me2.oclType = LocalExecution and me2.isRun = true implies (
      mes.oclType = ThreadStart  and  mes.signature = me2.statement
      and mes.name = me2.name
    )
    and  me2.oclType = ThreadComm implies (
      mes.oclType = Signal  and  mes.name = me2.arguments->at(1).name
    }
  ) // Scenario::Message.allInstances->exists
)
```

Figure 16 – Incomplete mapping of `Trace::MethodExecution` instances to `Scenario::Message` instances[5,6]

5.2 Identifying `followingMessage` links

The identification of the following messages of a given message (association `followingMessage` in the scenario metamodel in Figure 3), is the purpose of a separate rule. Recall that the self association `followingMessage` on `Scenario::Message` specifies the ordered sequence of messages that are triggered by a given message.

This is the purpose of the consistency rule shown in Figure 17. It is the conjunction of two OCL expressions. The first one 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

[6] Note that `LocalCall()` is not involved in this rule. Indeed, it is perfectly legal for a `LocalExecution` instance to have no `callee`.

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 (m1.destClassifier) is m2 source classifier (m2.sourceClassifier), m2 is sent after m1 is received (m1.timestampDest < m2.timestampSource) and there is no other Message instance sent to that classifier between those two timestamps.

The second conjunction checks that the elements of Sequence m.followingMessage, for any Message instance m, are sorted according to their timestamps (timestampSource).

```
Scenario::Message.allInstances->forAll( m1: Message, m2: Message |
  (   m1.destClassifier = m2.sourceClassifier
      and
      m1.timestampDest < m2.timestampSource
      and
      Scenario::Message.allInstance->select( m: Message |
          m.destClassifier = m1.destClassifier
          and
          m.timestampDest > m1.timestampDest
          and
          m.timestampDest < m2.timestampSource
      )->isEmpty
  ) implies m1.followingMessage->includes(m2)
)
and
Scenario::Message.allInstances->forAll(m: Message |
  Sequence{1..m.followingMessage->size}->forAll(i: Integer, j: Integer |
      i > j implies
                m.followingMessage->at(i).timestampSource
                    > m.followingMessage->at(j).timestampSource
)
```

Figure 17 – Identifying followingMessage links between Message instances

5.3 Identifying repetitions of Message instances

The last consistency rule matches instances of class `Trace::Repetition` and instances of class `Scenario::Repetition` (Figure 18). In its first three lines, the rule states that any instance of `Scenario::Repetition` corresponds to an instance of `Trace::Repetition` that is associated with `MethodExecution` instances. Indeed, a `Scenario::Repetition` instance is associated with `Message` instances, and the `Trace::Repetition` instance must thus involve the `MethodExecution` instances that correspond to these `Messages`.

The rest of the rule describes how the `Trace::Repetition` and `Scenario::Repetition` instances relate to each other, that is, how their attributes and links relate to each other. First, the kind of repetition, the clause, and the possible guard condition under which the repetition occurs must match (recall the distinction between the two associations relating classes `Scenario::Repetition` and `Scenario::Condition`).

```
Trace::Repetition.allInstances->forAll(Trep: Trace::Repetition |
  Trep.triggers->notEmpty implies
  Scenario::Repetition.allInstances->exists(Srep: Scenario::Repetition |
    //compare attributes
    Srep.kind = Trep.kind  and  Srep.forLoopVar = Trep.forLoopVar
    and  Srep.forLoopInc = Trep.forLoopInc  and  Srep.timeStamp = Trep.timeStamp
    and  Srep.clause.clause = Trep.clause.clause  //compare clause
    //compare conditions (compare whole sequences)
    and  Trep.getConditions().clause = Srep.guard.clause
    //compare Messages/MethodExecutions in repetitions
    and  Trep.triggers->forAll(me: MethodExecution |
           Srep->includesAll(CheckMapping.getMessages(me))
         )
  )
)
```

Figure 18 – Mapping Trace::Repetition to Scenario::Repetition

Last, the Message instances associated with the Scenario::Repetition instance must match the MethodExecution instances associated with the Trace:Repetition instance. This is checked using query operation getMessage(me:MethodExecution), which postcondition can be found in Figure 19. The following four different cases have to be considered:

- MethodExecution instance me, in the Trace::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.

- MethodExecution instance me is a RemoteCall[7]. In this case, getMessage(me) returns the (unique) Message instance that corresponds to me, using me's context and timestamp and the context and timestamp of the RemoteMethodExecution corresponding to me (using operation getRemoteMethodExecution() in Figure 20). The operation getRemoteMethodExecution() matches the RemoteCall to the corresponding RemoteMethodExecution as described in section 4.5.

- MethodExecution instance me is a StartThreadCall[8]. In this case, getMessage(me) returns the (unique) Message instance that corresponds to me, using me's context and timestamp and the context and timestamp of the

[7] 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.

[8] Note that LocalExecution instances with attribute isRun equal to true cannot be triggered by Repetition instances since the run() method of threads is automatically executed by the Java Virtual Machine (not the SUS source code).

`LocalExecution` instance (with attribute `isRun` equal to true) corresponding to `me` (using operation `getRunExecution()` in Figure 20). The operation `getRunExecution()` retruns the the `run()` method (a `LocalExecution` instance) that matches the `StartThreadCall` instance.

- `MethodExecution` instance `me` is a `ThreadComm`. In this case, `getMessage(me)` returns `Message` instances in which `me` is involved (using operation `getThreadComm()` in Figure 20). Recall that one write may correspond to several reads, so more than one `Message` instance can be returned.

```
CheckMapping::getMessages(me:MethodExecution):Sequence(Message)
post:
    me.oclType = 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.timestamp
        )->asSequence
    and
    me.oclType = RemoteCall implies
    result = Message.allInstances->select(m:Message|
        mapContextClassifier( CheckMapping.getRemoteMethodExecution(me).context,
                              m.destClassifier)
        and  mapContextClassifier(me.context, m.sourceClassifier)
        and  m.timestampDest = getRemoteMethodExecution(me).timestamp
        and  m.timestampSource = me.timestamp
        )->asSequence
    and
    me.oclType = StartThreadCall implies
    result = Message.allInstances->select(m:Message|
        mapContextClassifier(CheckMapping.getRunExecution(me).context,
                              m.destClassifier)
        and  mapContextClassifier(me.context, m.sourceClassifier)
        and  m.timestampDest = getRunExecution(me).timestamp
        and  m.timestampSource = me.timestamp
        )->asSequence
    and
    (me.oclType = ThreadComm and me.kind=#write) implies
    result = Message.allInstances->select(m:Message|
        CheckMapping.getThreadComm->exists(tcr:ThreadComm|
            mapContextClassifier(tcr.context, m.destClassifier)
            and  m.timestampDest = tcr.timestamp
            )
        and  mapContextClassifier(me.context, m.sourceClassifier)
        and  m.timestampSource = me.timestamp
        )->asSequence
```

Figure 19 – Postcondition of operation `CheckMapping::getMessages()`

```
CheckMapping::getRemoteMethodExecution(rc:RemoteCall):RemoteMethodExecution
post:
    result = RemoteMethodExecution.allInstances->select( rme |
        rme.clientThreadID = rc.threadID  and  rme.nodeID = rc.serverNodeID
        and  rme.clientNodeID = rc.nodeID
        and
        if rc.context.oclType = Trace::Instance then (
            rc.context.objectID = rme.clientObjectID
            and  rc.context.ofClass = rme.clientClassID
        ) else (
            rc.context.classID = rme.clientClassID
        ) endif
    )->asSequence->at(1)
```

```
CheckMapping::getRunExecution(stc:StartThreadCall):LocalExecution
post:
    result = LocalExecution.allInstances->select(le:LocalExecution|
        le.isRun = true
        and  stc.threadClassName = le.context.ofClass.name
        and  stc.threadClassID = le.context.ofClass.classID
        and  stc.threadObjectID = le.context.objectID
    )->asSequence->at(1)
```

```
CheckMapping::getThreadComm(tc:ThreadComm):Set(ThreadComm)
post:
    tc.kind = #write implies
    result = ThreadComm.allInstances->select(p:ThreadComm|
        p.kind = #read  and  p.context = tc.context
        and  p.returnValue = tc.argument->at(1)
    )
```

Figure 20 – Postconditions of operations `getRemoteMethodExecution()`, `getRunExecution()` and `getThreadComm()` of class `CheckMapping`

CHAPTER 6 INSTRUMENTATION

In order to produce an execution trace, a mechanism to instrument the code is necessary. The instrumentation must consist of adding statements that will produce traces (i.e., interact with the logging interface, described in Chapter 7). These statements must contain sufficient information about the current state of the system to be able to instantiate the trace meta-model as described in Chapter 4. The following subsection (6.1) describes what issues may be encountered when trying to accomplish this task. The second subsection evaluates various strategies to instrument code. Section 6.3 briefly presents our chosen strategy and section 6.4 explains how we used it for our tool.

6.1 Possible Instrumentation Issues

There are many factors that will influence the choice of the instrumentation strategy. Below we discuss certain issues that must be considered during the evaluation. Some issues will be solved in the discussions below, but others will be up to the chosen technique to resolve.

6.1.1 Basic Constructs

As described in Chapter 4, there are many events that need to be instrumented. Table 4 shows a brief summary of those elements and their required information (all must have a timestamp, thread identifier and node identifier):

Table 4 – Constructs to Instrument

Construct	Required Information
Method entry and exit	Method signature, executing object (or class), arguments
Condition beginning and end	Statement, clause
Repetition beginning and end	Statement, kind, clause
Calls to start threads	Class name, object identifier of the called thread
Calls to remote methods	Node identifier of called node
Execution of remote methods	Caller's thread and node identifiers
Thread communication	Collection identifier, kind (read or write), arguments

6.1.2 Destroy and Java

In object-oriented languages, the creation of an object can be tracked by instrumenting its class' constructor. Constructors can be instrumented much like other methods: by inserting instrumentation statements at the beginning and end of the constructor body.

Instrumenting the destruction of an object, on the other hand, may not be so straightforward. Some languages (such as Java) do not have explicit destructors. In this case, determining the point at which an object is destroyed may be quite complicated since it is not managed by the programmer.

In Java, one option is to add a `finalize()` method, if it does not already exist, to each class. This operation can then be instrumented. The `finalize()` method is called by the garbage collector on an object when garbage collection determines that there are no more

references to the object. Unfortunately, it may not be called as soon as an object's last reference is de-referenced, so it may be difficult to display it as a message coming from another object. That is to say, other messages may occur before the garbage collector destroys the object (and calls the `finalize()` method). The originator of the destruction will be hard to identify. Another option would be to keep count of the number of references to objects and instrument assignments. If the last reference is reassigned to null or to another object, this can be considered the destruction of this object. This may prove difficult and expensive because each assignment in the code must be tracked and evaluated to determine if an object looses its last reference.

This solution is quite complicated, especially as compared to instrumenting destructors in other languages. It raises a question: when a language doesn't let the programmer explicitly destroy objects, is it necessary to show destruction in sequence diagram? Since our ultimate goal is to compare generated sequence diagrams to the specification's sequence diagrams, it is important to consider what elements the designers have included in the specification. Though it is true that the implementation language may not have been chosen at that time, and so destruction of objects may be present in the documentation, it may not be a priority to implement when using a language that has memory management taken care of for the programmer, such as in Java. We consider instrumenting the destruction of objects very important when memory allocation is an issue and possibly a source of bugs, such as in C++. In Java, where memory allocation is mostly uncontrollable by the programmer, we consider the destruction of objects not pertinent enough and can be left out of the sequence diagram.

6.1.3 Remote Communication

In section 4.5 we have demonstrated that it is imperative that during remote communications, the client must know exactly which server it is calling and that the server must know exactly which client it is serving. In addition, it is not only the remote method executions that must be instrumented, but also all calls to remote methods. These might prove difficult to instrument for the following reasons.

Instrumenting calls to remote methods may be quite complex. Method calls are scattered throughout the code as opposed to method executions, where the beginning and end of a method is much more structured. Identifying remote calls may pose a problem as well; only calls on an object that implements an interface that extends the `Remote` interface need to be instrumented, not all method calls. This may require static analysis to store the names of the classes that implement those interfaces.

Yet another potential problem is that the server needs information about the client that has invoked it. In Java, server objects only have access to a client's IP address. This is not sufficient; there may be many virtual machines running on the client computer, thus all will have the same IP address. Moreover, many threads may be invoking remote methods. There obviously exist mechanisms in RMI to uniquely map calls to execution, but they are not available to the programmer. Therefore, servers need to know a client's node identifier and thread identifier, which it does not have direct access to. It is up to the instrumentation strategy to implement this (see section 6.1.4).

Furthermore, clients need to differentiate between servers. Clients may have references to many servers of the same type, but must have a mechanism to identify them. One

solution is to use the name the server is bound to in the registry. When a server can fulfil requests, it binds its name into the RMI registry. This name must be unique; the registry will return an error if this name is already taken. When a client wants a server stub, it looks up the server name in the registry to obtain it. Instrumentation could keep track of this name when the client performs a lookup in the registry. This approach has one major drawback: it does not take into account RMI callbacks and unnamed (or anonymous) remote objects. RMI callbacks are when a stub is sent as an argument to a remote method, allowing a client to become an RMI server.

These partial solutions are insufficient. A client needs a mechanism to send information about itself to a server and vice-versa. Our proposed solution to accomplish this is to add extra parameters to remote methods and to change the return type of these methods. This way, the client's information can be sent when invoking a remote method and the server's information can be return along with the original return value. This may involve more sophisticated instrumentation because all calls to remote methods must be intercepted to add the extra arguments. New methods must be written to match all remote methods to have additional parameters and a different return type that will allow the server to return its node identifier. This has the potential to be complex, depending on the instrumentation strategy.

6.1.4 Identifiers

Nodes, threads, objects and classes will all need to be uniquely identified since they will need to be distinguished from others of the same type in the traces. Each is explained in turn in the next sections.

6.1.4.1 Node Identifiers

As described in section 4.5, each node should have its own identifier to differentiate them from other nodes. Because each node must contact the logging server before the execution of the system (see section 7.2) a unique number for each node will be generated by the logging server. This identifier, which can be generated by a simple counter, would help in sorting the instrumentation statements once they are received by the logging server (section Chapter 7). Note that, as mentioned above, using an IP address for the node identifier is not adequate because it is not a unique identifier when there are many virtual machines on the same computer.

6.1.4.2 Thread Identifier

Like the node identifier, each thread needs to be uniquely identified. Java threads do not have an identifier built into the language, except for the `name` attribute, which may not be unique and can be changed. Therefore, thread identifiers must be generated. Because `Thread` is a library class, we will assume that an identifier attribute cannot be added. We propose the following approach: thread objects can be placed in a hash table where the key is the actual thread object, and the corresponding value is a unique number. When one needs the identifier of a thread, the hash table is looked up. If the thread object is found in the table, the identifier is returned. If it is not in the table, a new entry is created with the thread object as the key and the next available identifier as the value, which is returned. The next time the table is queried with this thread, that same identifier will be returned. The values assigned to each thread must be unique in that particular node, but

two threads on different nodes can have the same thread identifier. The combination of the threadID and the nodeID uniquely identifies a thread.

6.1.4.3 Class Identifiers

In Java, classes cannot have the same name within the same package. Therefore, class names combined with their package names are unique and are used as class identifiers. The convention we will use is the same as in the language: package names in hierarchical order, separated by ".", followed by the class name (ex: java.util.ArrayList).

In the scenario metamodel, the class `Class` has the attribute `classID`. It is purely for displaying the resulting scenario diagram as the class name can become quite long. The same class on different nodes have the same identifier.

6.1.4.4 Object Identifiers

Since sequence diagrams describe messages between objects, uniquely identifying objects is essential. The literal name given to objects in the program is not sufficient because of aliasing. Also, the same variable name can be used in different contexts to identify different objects.

One solution to consider is to use the object's physical memory address. This would be unique to each object. This is the approach used in [6]. In our context, this is inadequate since the Java garbage collector may swap the objects around the memory space to eliminate fragmentation or to accomplish other maintenance tasks.

Another solution is to use Java's `hashcode()` construct. This is a method that is inherited from `object`, so every object has access to it. The java specification [36] specifies that `hashcode()` returns a hash code value for an object. Though it is often used as a unique identifier, Java specification does not require that this method return a unique integer for each object. It is recommended in the Java API that `hashcode()` return a unique integer since, if it does not, it renders a hash table retrieval into a linear search, but this is not a requirement nor is it enforced. Furthermore, two distinct objects in the same state (i.e. the `equals()` method returns true) should return the same hash code. For these reasons, the `hashcode()` method does not meet our requirements.

Our chosen solution is to include an identifier member variable in each object when instrumenting the code to count the created instances of the class. Its value is set in the class's constructor using a static member that holds the next object identifier and is incremented each time an object is created.

Different objects of a same class in a node have different identifiers. Different objects of different classes on different nodes can have the same identifier. Objects are uniquely identified by the combination of the node identifier, the class identifier and the object identifier.

6.2 Instrumentation Techniques: A Survey

The primary goal of this section is to determine a strategy to instrument the code in order to produce trace statements. Possible strategies are evaluated based on the feasibility, straightforwardness and completeness of the solution (i.e., does it solve the issues above).

Execution time, intrusiveness and multiple versions of the code are other factors taken into account.

The tool that implements the instrumentation must minimize the time of execution. The time the tool takes to analyze the target system's code will be understandably proportional to the amount of code, but must still complete in a reasonable amount of time. Once the system is instrumented, it will be run. Although the system will be larger, the instrumentation should not augment the execution time by a significant amount. This is related to the intrusiveness of the instrumentation. Instrumentation statements should not take a long time to run, but also must not introduce delays that might change the normal order of operation of the system. In other words, the tool should seek to minimize the "probe effect", the interference it has on the original system [33]. Another important aspect is to avoid having two versions of the code. Ideally the instrumentation statements will not be inserted directly into the target system's code but will instead be kept separate. If the instrumentation phase transforms the code, there will be two versions. This can lead to inconsistencies if the original target code is changed. Also, the instrumentation statements should not be presented to the developer or tester. All these factors will help determine the most desirable solution to each problem.

The following sections present the tools and languages that could assist in instrumenting the code to produce the instrumentation statements.

6.2.1 Perl

Perl is a powerful scripting language that would lend itself well to instrumenting code. Its regular expression set, which is vital for pattern matching, is unparalleled. It also

effortlessly parses files, without restrictions on buffer or string length. File manipulation is easily accomplished, as Perl can create, save, copy, move and rename files, just to name a few operations. In order to use Perl for instrumentation, a program must be written that reads in a source file and that matches each statement to the constructs that need to be instrumented, as described above. If there is a need for an instrumentation statement, the program inserts the appropriate statement at that location. The statement inserted must interact with the logging interface. It must be noted that such an endeavour might prove difficult due to the many ways of coding. For example, in many programming languages there are several possible places for an opening brace "{": on the same line as the previous statement or the next line, indented or not, etc. Perl has been used in other research papers to instrument code [6], [15] and can be used to instrument any language. However, Perl does not provide a way to easily detect method calls. In addition, using this approach will result in two versions of the code: the original code and another with the instrumentation statements.

6.2.2 JVMPI

The Java Virtual Machine Profiler Interface (JVMPI) [37] was created by Sun as an interface to be used by tool vendors wanting to develop profilers. The virtual machine notifies a profiler agent of events such as heap allocations and method entry. Then, the profiler agent issues controls and requests for more information through the JVMPI. For example, the profiler agent can turn on/off a specific event notification, based on the needs of the profiler (developed by the tool vendor).

A few of the available events would be practical for our purposes. Method entry and exit are obvious events that are needed. Thread start and end could also be useful. Unfortunately, there are no events marking control flow. Conditions and iterations do not trigger JVMPI events. This method cannot be used as an instrumentation mechanism without being paired with another method. Additionally, JVMPI is a Java specific solution. Since it utilizes the virtual machine, there are no corresponding tools in most other programming languages.

6.2.3 Abstract Syntax Trees

Another method to instrument code is to use Abstract Syntax Trees (ASTs). Once the tree representing the code is built, instrumentation statements can be inserted and then the modified tree can be reconstructed into code. Two tools, ANother Tool for Language Recognition (ANTLR) [1], and JavaCC [17], are language tools that provide a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, or C++ actions. Since a Java grammar is available for both, it would be possible to use one of the tools to generate an AST of the Java code. Then, the instrumentation statements could be inserted by parsing the tree and finding the desired statements to instrument. It is then possible to rewrite the code from the new tree. The biggest disadvantage of this approach is that it results in two versions of the source code.

6.2.4 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a paradigm that has two essential goals: to allow for the separation of cross-cutting concerns and to provide a mechanism for the description of these concerns. AspectJ [2] is an implementation of AOP for the Java

language built as an extension to the language. AspectJ allows the programmer to define join points, points in the program that are of interest such as a method entry, and execute code if such a statement occurs. This may be used to identify method entry and exit, as well as field access. When one of these events occurs, an instrumentation statement could be executed. This approach has the significant advantage of a separation of the target source code and the instrumentation code. Like JVMPI (section 6.2.2), this method does not provide means to extract control flow information and so cannot be used as an instrumentation mechanism without being paired with another method. AspectJ is a Java specific solution, but there are comparable AOP implementations for most other languages, such as AspectC++ for C++ and AspectS for SmallTalk, so we will consider aspect-oriented programming as language independent.

There are other AOP implementations for Java, such as JBossAOP, Nanning, Aspectwerks, EAOP, JAC, HyperJ and CeasarJ. As far as we know, none of these implement all of AspectJ's functionality. They are more recent and so have not evolved as far as AspectJ. Most of them only allow method call and execution interception. Some do have the advantage that aspects are written in Java, and not an extension of the language, but this is not a significant advantage for them to be considered.

6.2.5 OpenJava

OpenJava is a macro system for Java. It allows manipulation of a program at compile time. It uses metaobjects (as opposed to ASTs) to abstract logical entities of a program. This abstraction allows the macro programs to access the data structures representing the logical structure of the program. In other words, this approach is very similar to ASTs, in

that statements can be added in the required places in the structure, but provides a much more elegant interface to accomplish this. The macros are then processed to weave the code into the system.

Though the macro program can be separated from the target source code, each class in the SUS must contain an additional statement to allow it to instantiate a metaobject. Additionally, there is no simple mechanism to add control flow information without searching each statement in every method.

6.2.6 Summary

Table 5 shows a summary of the proposed instrumentation strategies. From this comparison, we can see that Perl and AST, which have the same advantages and disadvantages, and JVMPI do not meet enough of our needs. Among the choices, Aspect Oriented Programming and OpenJava are appealing; however AOP has the most desirable advantages. Clear separation between target code and instrumentation is a substantial asset, as is automatic detection of method calls. Its major disadvantage, lack of control flow information, has been identified as a possible addition to future releases of AspectJ. The next section is a brief introduction to AOP and AspectJ.

Table 5 – Comparison of Instrumentation Techniques

	Perl	**JVMPI**	**ASTs**	**AOP**	**OpenJava**
Different version of the Java source code	Yes	No	Yes	No	Yes
Language Independent	Yes	No	Yes	Yes	No

Provides Control Flow Information (but requires to search for them)	Yes	No	Yes	No	Yes
Needs Separate Static Analysis	No	Yes	No	Yes	Yes
Detects Method Calls (without looking through each statement)	No	No	No	Yes	No
Allows Adding New Methods (with new parameters) and Fields to Classes	Yes	No	Yes	Yes	Yes

6.3 Aspect Oriented Programming and AspectJ

Aspect-oriented programming [10] is a recent methodology that facilitates the modularization of concerns in software development. In particular, it extracts scattered concerns from classes and turns them into first-class elements: aspects. By decoupling these concerns and placing them in aspects, the original classes are relieved of the burden of managing functionalities orthogonally related to their purpose. Later, the aspect code is injected into appropriate places by a process known as weaving. Aspects contain *join points* that specify well-defined execution "points" in the execution of the instrumented program where aspect code interacts, e.g., a specific method call or execution. *Pointcuts* describe sets of join points by specifying, for example, the objects and methods to be considered. An *advice* is additional code that should execute before or after join points. It can even have control on whether the join point can run at all.

A direct consequence of aspect use is that less code needs to be written, code that would otherwise be spread throughout the system can now be localized in one place. By keeping

aspects separate from the SUS methods they interact with, the SUS source code is more maintainable and easier to understand.

This work uses AspectJ (AOP for Java), a well-known implementation of AOP. The general structure of an AspectJ aspect can be found in Figure 21. The first line of the aspect, following the AspectJ syntax [13], specifies its signature, which starts with one of the three possible types of advice, namely *before* (used to execute some code right before the point cut), *after* (used to execute some code right after the point cut), and *around* (used to execute some code before and after the point cut): Figure 21 specifies an around advice. The point cut declaration follows the advice type. Different functionalities are available to specify the point cut. For instance it is possible, using the `execution` and `within` functionalities, to specify a point cut as any execution of specific method name, say `getName()`, in any class in any package, except in a given package, say `somePackage`, as in Figure 21. Other such functionalities can be found in [13]. Following the point cut declaration is the advice proper, that is, what has to be done before, after or around the execution of the point cut. In the around case, one can decide to execute the original point cut using AspectJ statement `proceed()`, as in Figure 21, and execute whatever Java statements are deemed necessary before and after (in case of an around advice) this execution. One can even decide not to execute the original point cut. The reader interested in more technical details on AspectJ is referred to [13].

```
around(): execution(* * *.getName()) && !within(* * somePackage)
{
    // any Java statement
    proceed();
    // any Java statement
}
```

Figure 21 – General structure of an AspectJ aspect, an example

6.4 AspectJ Templates

We illustrate below our use of AspectJ by means of four examples: adding identifiers (section 6.4.1), intercepting constructor and (static) method executions (section 6.4.2), intercepting RMI communications (section 6.4.3) and intercepting thread communications (section 6.4.4). A complete list of aspect templates, including templates related to the interception of control flow structures, can be found in Appendix B.

Each time, instead of an aspect, which would be specific to a particular SUS, we show aspect code templates. These templates are generic descriptions of aspects showing what parts vary according to the SUS: the varying parts will be written in bold face. Aspects use utility classes that are provided in an instrumentation package where the aspects are also located.

The main goal of the aspects is to produce trace statements. This is handled by the class `LoggingClient`, presented in section 7.1. When producing a trace statement, `LoggingClient`'s `instrument(List, Object[])` method is called. It has two parameters: the first is a `List` of `Strings` that contains all of the elements of the trace statement and the second is an array of `Objects` containing arguments, if there are any. The `instrument(List, Object[])` method automatically reports on the timestamp, thread identifier and node identifier, so these do not need to be sent by the aspect as arguments to instrument. It also implements the way the trace statements appear and are recoded, abstracting these from the aspects. In our case, the trace statements are written

to a file according to its thread identifier. Section 7.1 contains the implementation of this method.

6.4.1 Adding Identifiers

First, AspectJ is used to add attributes, methods, and interfaces to classes in the SUS so that, during SUS methods executions, unique identifiers required by the trace metamodel (Figure 8) are correctly set and available to other aspects: SUS class instances are uniquely identified by an object identifier and their class name (or `classID`).

The AspectJ code to accomplish this is shown in Figure 22, which must be repeated for all classes that need to be instrumented in the SUS. The first statement adds the field `objectID` to the class `ClassName`; it is initialized by the method `objectIDgenerator()`. The next two statements add to the class `ClassName` a static field `currentObjectID` (initially set to zero) and a static method `objectIDgenerator()`, which generates a new identifier each time the method is called. Therefore, each time a new instance of `ClassName` is created, the field `objectID` is initialized to a unique identifier. This identifier is unique only for a given class; it must be combined with its class name (or `classID`) to be unique relative to all other objects in the node.

The `declare parents` statement adds the interface `ObjectID` (defined last in Figure 22) to `ClassName`. The method `getObjectID()` is implemented because it is now required by the interface. This interface is needed to be able to cast objects when needing the `objectID`.

```
public int ClassName.objectID = ClassName.objectIDgenerator();

private static int ClassName.currentObjectID = 0;

private static int ClassName.objectIDgenerator() {
  return ClassName.currentObjectID++;
}

declare parents : ClassName implements ObjectID;

public int ClassName.getObjectID() {
  return objectID;
}
```

```
public interface ObjectID {
  public int getObjectID();
}
```

Figure 22 – Instance identifier objectID

6.4.2 Intercepting Constructor and Method Executions

In order to instantiate class `LocalExecution` in the trace metamodel (Figure 8), it is important to intercept any execution of any method in the SUS. Since the information we have to manipulate (i.e., retrieve from the SUS execution and use to instantiate the trace metamodel) is different for a constructor, a static method and a non-static method, we devise three aspect templates, as discussed in this section.

The first template (Figure 23) is for tracing constructor executions. It specifies an around advice as we want to produce trace information before and after the execution of the constructors. Indeed, we want to detect when (i.e., at which moments in time) constructors start and end executing as this will tell us what are their nested statements (Figure 8): the statements within the same thread (uniquely identified by the `threadID`) between the two `timestamps` reported at constructor start and end. The point cut specified in Figure 23 is any execution of method `new` (AspectJ notation to indicate

constructors) with any number of parameters (`new(..)`), on any class in a given package[9] that is in the SUS (this also includes sub-packages and inner classes). Note that any around advice must return an instance of type `Object` (`Object around():...`). This is not useful for constructors so we return `null` (last statement of the advice). The point cut also prevents any tracing of the aspects themselves, assuming aspect classes are in a specific package, namely `Instrumentation` (`!within(Instrumentation.*)`).

The around advice then consists of three parts: tracing before the execution of the constructor, proceeding with the execution of the constructor and tracing after the execution of the constructor.

Before the execution of the constructor the advice reports on:

- The type of pointcut (e.g., the execution start of a constructor labeled as `"Create method start"`);
- The name of the class for which an instance is to be created;
- The signature of the constructor, which is obtained by using AspectJ's reflection facilities.

The thread identifier, node identifier and timestamp are taken care by the `instrument(List, Object[])` method, so these are not included in the list of strings.

The execution of the (intercepted) constructor is achieved using AspectJ statement `proceed()`. After the execution of the constructor, but before the end of the advice, the advice reports on: the end of the constructor execution (`"Create method end"`) and the

[9] Specifying multiple packages or classes can be performed as follows: `( execution(PackageName1..* new(..)) || execution(PackageName2..* new(..)) || execution(PackageName3.className new(..)) ) && !within(Instrumentation.*).`

unique object identifier of the newly created object. Note that in order to get the `objectID`, the reference of the created object (obtained in AspectJ with `thisJoinPoint.getThis()`) is casted to `ObjectID` (see section 6.4.1).

As a result, given a trace statement reporting on the execution of class A's constructor at timestamp t1 and a trace statement reporting on the end of this constructor at timestamp t2, any trace statement with the same thread and node identifiers as the two above statements with a timestamp between t1 and t2 is executed by the constructor. This is how nested statements are identified and ordered in the trace metamodel (Figure 8).

```
Object around(): execution(PackageName..*.new(..))
                 && !within(Instrumentation.*)
{
  ArrayList log = new ArrayList();

  log.add("Create method start");
  log.add(thisJoinPointStaticPart.getSourceLocation().getWithinType().getName());
  log.add(thisJoinPointStaticPart.getSignature().toLongString());
  LoggingClient.getLoggingClient().instrument(log, thisJoinPoint.getArgs());

  proceed();

  log.clear();
  log.add("Create method end");
  log.add(String.valueOf(((ObjectID) thisJoinPoint.getThis()).getObjectID()));
  LoggingClient.getLoggingClient().instrument(log, null);

  return null;
}
```

Figure 23 – Aspect template for tracing constructor executions

The next two templates are very similar to the first one. The template in Figure 24 is for tracing any static method executions (thus the keyword static in the pointcut declaration). The main difference is that after the execution of the static method, no object identifier is

reported, but information on the possible returned value is (the second parameter to the `instrument(List, Object[])` method).

The template in Figure 25 is for tracing non-static method executions that are not `run()` methods in threads: Intercepting executions of `run()` methods in threads is the focus of Section 6.4.4. What is new in the pointcut declaration is the parameter to the pointcut named `self`. This parameter allows the advice to use keyword `self` as a reference to the object on which the intercepted method is executed and prevents this pointcut from catching static methods (`this(self)` is `null`). Consequently, the advice reports on the `self` object unique identifier before the intercepted method execution.

```
Object around(): execution(static * PackageName..*.*(..))
                 && !within(Instrumentation.*)
{
  ArrayList log = new ArrayList();

  log.add("Static method start");
  log.add(thisJoinPointStaticPart.getSourceLocation().getWithinType().getName())
  ;
  log.add(thisJoinPointStaticPart.getSignature().toLongString());
  log.add(thisJoinPointStaticPart.getSignature().getName());
  LoggingClient.getLoggingClient().instrument(log, thisJoinPoint.getArgs());

  Object[] returnArg = {proceed()};

  log.clear();
  log.add("Static method end");
  LoggingClient.getLoggingClient().instrument(log, returnArg);

  return returnArg[0];
}
```

Figure 24 – Aspect template for tracing static method executions

```
Object around(Object self): execution(* PackageName..*.*(..))
                            && !within(Instrumentation.*)
                            && !execution(void Runnable+.run(..))
                            && this(self)
{
  ArrayList log = new ArrayList();

  log.add("Method start");
  log.add(String.valueOf(((ObjectID) self).getObjectID()));
  log.add(self.getClass().getName());
  log.add(thisJoinPointStaticPart.getSignature().toLongString());
  log.add(thisJoinPointStaticPart.getSignature().getName());
  LoggingClient.getLoggingClient().instrument(log, thisJoinPoint.getArgs());

  Object[] returnArg = {proceed(self)};

  log.clear();
  log.add("Method end");
  LoggingClient.getLoggingClient().instrument(log, returnArg);

  return returnArg[0];
}
```

Figure 25 – Aspect template for tracing non-static method executions

6.4.3 Intercepting RMI Communications

Using RMI in Java, a call to a remote method (i.e., a call to an object whose class implements `java.rmi.Remote`), is very similar to a call to a local object. For instance, RMI provides synchronous communication, i.e., when the RMI client invokes a remote method at the server, the client itself is blocked until the method invoked returns. In addition, the remote method executing at the server does not have any information on the client it is serving. As a consequence, since the call and the execution occur on two different nodes in the network, we cannot rely on the conjunction of node identifiers, thread identifiers and timestamps to order trace information. We thus have to intercept executions of methods in classes implementing interface `java.rmi.Remote` on the server side, and calls to methods on objects which class implements interface `java.rmi.Remote` on the client side.

6.4.3.1 Server side

Additionally, we have to gather enough information when intercepting these calls/executions to match a call on the client side with an execution on the server side. We will see below that information thus has to be passed and returned along with the call. In order to do so, we create aspects that wrap around methods in `java.rmi.Remote` interfaces on the server side. For each such remote method, named for instance `myRemoteMethod(...)`, the aspect adds method `myRemoteMethodExtra(...)` (with its body) to the interface[10]. This added method has one additional parameter that is used to pass information about the client performing the call: i.e., `client threadID, nodeID, className, objectID` (recall class `RemoteMethodExecution` in the trace metamodel). It also has a different returned value to pass information on the server back to the client: the `serverNodeID` (Figure 8). The added parameter containing client information is used by `myRemoteMethodExtra(...)` to produce the trace on the server side, and `myRemoteMethodExtra(...)` then calls the original method `myRemoteMethod(...)` with the original parameters. This will result into a `caller-callee` relationship between an instance of `RemoteMethodExecution` (corresponding to the execution of the added method) and an instance of `LocalExecution` (corresponding to the execution of the original method).

The corresponding aspect template is shown in Figure 26. A method is added to the remote interface `InterfaceName`, and the original method name (`MethodName` in the template) is used to name that added method (i.e., `MethodNameExtra`). The added method

[10] Note that AspectJ allows the addition of concrete methods with bodies to interfaces, though this is not allowed by Java.

has one first parameter (`client`) of type `UniqueID`, followed by the parameters of the original method. Class `UniqueID` is provided by the instrumentation package, and has attributes to store the information that has to be sent to the server (the identifiers mentioned above). Note that this aspect is different than the previous ones as no pointcut is defined: We do not intercept anything but only add a method: as discussed below, we then need to intercept calls to the original method, on the client side, and transform them into calls to the added method.

Before calling the original method, the aspect produces a trace statement in which are reported: The start of a remote execution (labeled `"Remote method execution start"`) and the `objectID` and `className` for the object executing the call on the server side. Additionally, information on the client is provided: Client `threadID`, `nodeID`, `className`, and `objectID`. When reading trace statements, the first identifiers will be used to initialize attributes defined in `ExecutionStatement` and to initialize a `context` for the execution, and the other identifiers will be used to initialize attributes in class `RemoteMethodExecution` (Figure 8). Note that the execution of the original method will be intercepted by the aspects introduced previously. The added method then calls the original one and stores the result in a data structure (ArrayList) that is eventually returned (instead of the original return value). The second element of the data structure stores the `nodeID` of the server. The bundling of the original return value with the `nodeID` is how the information about the server is transmitted to the client. Before returning this `ArrayList` instance to the client, another trace statement is produced, indicating the end of the remote method execution.

Note that the aspect in Figure 26 is for remote methods that have a return value. Another, very similar aspect for method without any return value can be found in Appendix B.

```
public Object  PackageName.InterfaceName.MethodNameExtra(
                    UniqueID client
                    [, parameters of the method - if any]
               ) throws RemoteException [, other throws clauses - if any]
{
  ArrayList log = new ArrayList();

  log.add("Remote method execution start");
  log.add(String.valueOf(((ObjectID) this).getObjectID()));
  log.add(this.getClass().getName());
  log.add(String.valueOf(client.threadID));
  log.add(String.valueOf(client.nodeID));
  log.add(client.className);
  log.add(String.valueOf(client.objectID));
  LoggingClient.getLoggingClient().instrument(log, null);

  ArrayList retArray = new ArrayList(2);
  retArray.add(0, MethodName([arguments of the method - if any]));
  retArray.add(1, new Integer(LoggingClient.getLoggingClient().getNodeID()));

  log.clear();
  log.add("Remote method execution end");
  LoggingClient.getLoggingClient().instrument(log, null);

  return retArray;
}
```

Figure 26 – Aspect template for tracing execution of remote methods

6.4.3.2 Client side

On the client side, any call to a remote method has to be intercepted, to instead perform a call to "Extra" remote methods, add the required first parameter, and analyze the returned `ArrayList` instance. This is the purpose of the aspect template in Figure 27. This is an around advice intercepting any call to remote methods (`call(* java.rmi.Remote+.*(..))`). The signature also allows the advice to access the object on which the call is performed using reference name `targetObj` (`target(targetObj)`). Note that we need an around advice here, as we have to change the call, and only around

advices allow that. Additionally, we have to get the `serverNodeID` from the returned `ArrayList`.

Note that the first statement in the advice tests whether the class name of the object on which the call is performed ends with string `"_Stub"`, i.e., whether the object is a RMI stub representing a remote object. This is necessary as the point cut also specifies calls, local to the server, to methods in `Remote` interfaces. These are local calls and should not result in `RemoteCall` trace statements: In such a case, the advice does nothing but to proceed with the execution, without any trace statement produced (`return proceed(targetObj)` at the very end of the aspect).

The first thing performed by the aspect is to produce a trace statement: A `"Remote method call start"`. 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. The rest of the aspect template prepares and performs the call to the "Extra" method added to the remote interface. This is performed thanks to reflection mechanisms available in Java, and we do not further describe these statements here. The returned `ArrayList` is then used to (1) get the node identifier of the server (second element of the `ArrayList`) and (2) return the result of the original remote method (first element of the `ArrayList`).

As discussed in Section 6.4.2, at a node in the network (i.e., given a `nodeID`), trace statements can be ordered within a thread (i.e., given a `threadID`) using `timestamps`, and this is the way `nestedStatements` are determined and ordered. In the case of a distributed system, we also have to order calls to remote methods and their matching

remote method executions. As described in Section 4.5, this is done using (1) the data we send along with any remote call (including `nodeIDs`, `threadIDs`, and `timestamps`), (2) the data returned by the call (i.e., the server node identifier), and (3) the fact that RMI provides synchronous communications, and there is therefore no need for a global clock.

```
Object around(Object targetObj): call(* java.rmi.Remote+.*(..))
                                 && target(targetObj)
                                 && !within(Instrumentation.*)
{
if (targetObj.getClass().getName().endsWith("_Stub")) {

  ArrayList log = new ArrayList();

  log.add("Remote method call start");
  LoggingClient.getLoggingClient().instrument(log, null);

  //call same method but with "Extra" appended to the name, with extra parameter
  Class targetClass = targetObj.getClass();
  Signature signature = thisJoinPointStaticPart.getSignature();
  Object thisObj = thisJoinPoint.getThis();

  Object[] arguments = thisJoinPoint.getArgs();
  Class[] argTypes = ((CodeSignature)signature).getParameterTypes();
  Object[] newArgs = new Object[arguments.length + 1];
  Class[] newArgTypes = new Class[arguments.length + 1];

  if(thisObj == null) { //the call is performed in a static method
    newArgs[0] = LoggingClient.getLoggingClient().getUniqueID(
        thisJoinPointStaticPart.getSourceLocation().getWithinType().getName());
  }
  else { //the call is performed in a non-static method
    newArgs[0] = LoggingClient.getLoggingClient().getUniqueID(
        thisObj.getClass().getName(), ((ObjectID) thisObj).getObjectID());
  }

  newArgTypes[0] = Class.forName("Instrumentation.UniqueID");
  System.arraycopy(arguments, 0, newArgs, 1, arguments.length);
  System.arraycopy(argTypes, 0, newArgTypes, 1, argTypes.length);

  Method method = targetClass.getMethod(signature.getName() + "Extra",
                                        newArgTypes );

  //perform the call to the extra method
  ArrayList returnArray = (ArrayList)method.invoke(targetObj, newArgs);

  log.clear();
  log.add("Remote method call end");
  log.add(String.valueOf(returnArg.get(1)));
  LoggingClient.getLoggingClient().instrument(log, null);

  return returnArray.get(0);
} else {
  return proceed(targetObj);
}
```

Figure 27 – Aspect template for tracing calls to remote methods

6.4.4 Intercepting Thread communications

Thread communication can take the form of the start of a thread, that is a call to `start()` (that automatically triggers method `run()`), but also asynchronous message passing. The first two aspect templates we present in this section address the former kind of communication.

In order to identify who starts which thread, it is necessary to intercept any call to operation `start()` on objects whose class implements interface `java.lang.Runnable` (aspect in Figure 28), and any execution of operation `run()` by those objects (aspect in Figure 29). The first aspect contains a before advice (we do not need to change the call or produce any trace after it). The advice consists in reporting on the statement type (`"Start method call"`), the `objectID` of the called thread and its class name. This will allow a one-to-one mapping of the call and the method execution, as described in Section 4.4.2. On the other hand, the aspect in Figure 29 is an around advice (we want to identify statements executed by `run()`, i.e., between its start and its end using timestamps) and is very similar to the aspect for intercepting non-static method execution. The main differences are the pointcut definition which is specifically to the `run()` method, and the trace statement type: `"Run method start"`.

```
before(Object targetObj): call(void Runnable+.start())
                          && target(targetObj)
                          && !within(Instrumentation.*)
{
  ArrayList log = new ArrayList();

  log.add("Start method call");
  log.add(String.valueOf(((ObjectID) targetObj).getObjectID()));
  log.add(targetObj.getClass().getName());
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

Figure 28 – Aspect template for tracing calls to start on thread objects

```
Object around(Object self): execution(void Runnable+.run())
                           && this(self)
                           && !within(Instrumentation.*)
{
  ArrayList log = new ArrayList();

  log.add("Run method start")
  log.add(String.valueOf(((ObjectID) self).getObjectID()));
  log.add(self.getClass().getName());
  LoggingClient.getLoggingClient().instrument(log, null);

  Object returnArg = proceed(self);

  log.clear();
  log.add("Run method end");
  LoggingClient.getLoggingClient().instrument(log, null);

  return returnArg;
}
```

Figure 29 – Aspect template for tracing executions of `run()` methods

Regarding signal passing (see Section 4.4.3), we assume thread asynchronous communications are achieved by means of data structures that hold `Signal` objects: one thread writes an instance of a child class of abstract class `Signal` to the data structure, using a specific method, and another thread reads the data structure using another specific method. Furthermore, the information about those data structures and methods has to be provided by the user before the instrumentation. It would be indeed too expensive to instrument any data structure (and its methods) used in threads in a Java program. This would amount to instrumenting any class inheriting from `Collection` for instance.

Rather, the instrumentation phase uses a configuration file indicating which classes (along with "write" and "read" operations) could hold `Signal` instances, based on an organization's specific programming standards and practices, for instance. The aspect template in Figure 30 then describes the interception of calls to these methods. It is also an around advice as we want to gather information on the returned value (i.e., is it a signal object?). The point cut specifies that any call to method `methodName` in class `className` (i.e., a "write" or "read" method in a data structure specified in the configuration file) in package `PackageName` is intercepted. The advice only consists in reporting enough information in order to decide, offline, i.e., when analyzing the trace, whether or not the intercepted call is really involved into an asynchronous thread communication. In order to do so, the advice reports on the arguments and return value (their class and object identifiers). If the class is child class of `Signal`, then the aspect has intercepted part of an asynchronous thread communication.

```
Object around(Object dataStruct): call(PackageName.className.methodName(..))
                                && target(dataStruct)
                                && !within(Instrumentation.*)
{
  ArrayList log = new ArrayList();
  int coll = CollIDmap.getCollIDmap.getCollID(dataStruct);

  log.add("ThreadComm call start");
  log.add(String.valueOf(coll);
  log.add(thisJoinPointStaticPart.getSignature().toLongString());
  log.add(thisJoinPointStaticPart.getSignature().getName());
  LoggingClient.getLoggingClient().instrument(log, thisJoinPoint.getArgs());

  Object[] returnArg = {proceed(self)};

  log.clear();
  log.add("ThreadComm call end");
  LoggingClient.getLoggingClient().instrument(log, returnArg);

  return returnArg[0];
}
```

Figure 30 – Aspect template for tracing (possible) asynchronous communications

Note that the collection instance is associated with a unique identifier, using operation `getCollID()` in class `CollIDmap`, provided in the instrumentation package, where the aspect code lies (see Appendix B). This class associates unique identifiers with collection instances and as a result the aspect does not depend on whether the data structure is provided by the JDK or part of the SUS.

In the case where the data structure is not from the Java library, but is instead a class from the SUS, the method executions will be reported by the aspect presented in section 6.4.2. If one corresponds to a thread communication (whether "read" or "write"), it will be transformed from a method execution into a thread communication offline.

6.5 Instrumenting Control-Flow Structures

As stated at the beginning of Section 6.2.4, AspectJ does not currently provide mechanisms to intercept the control flow and, as a temporary measure, we have to resort to instrumenting the SUS source code. We defined a class within the aspect code, namely `TracingCTRLFlow`, that provides operations (with empty bodies) for every control flow structure in Java, e.g., it has operations `ifStatementStart()` and `ifStatementEnd()` for detecting the start and the end of an if statement[11] as illustrated in Figure 31. Calls to `TracingCTRLFlow` methods are inserted in the SUS source code at appropriate places to be intercepted for instrumentation purposes, e.g., at the beginning and end of the `if` and `else` blocks.

[11] Note that these two operations can also be used for the `else` part of an `if` statement (the call provides the negation of the `if` statement as a clause), and for `switch` statements.

```
public class TracingCTRLFlow {
  public static void ifStatementStart(  String Statement,
                                        String clause) {}
  public static void ifStatementEnd() {}

  public static void whileStatementStart(String Statement,
                                         String clause) {}
  public static void whileStatementEnd() {}

  public static void doWhileStatementStart() {}
  public static void doWhileStatementEnd(String Statement,
                                         String clause) {}

  public static void forStatementStart( String Statement,
                                        String var,
                                        String clause,
                                        String inc) {}
  public static void forStatementEnd() {}

  public static void breakStatement() {}
  public static void continueStatement() {}
}
```

Figure 31 – Class `TracingCTRLFlow` and its operations

Calls to these methods are then intercepted by specific aspects, reporting on the kind of control flow structure being executed, the context of the execution (`objectID`, `className`), and the clause driving the flow of control (plus the initialization and update parts of `for` loops). Figure 32 shows the aspects intercepting call to operations `ifStatementStart()` and `ifStatementEnd()`, and the complete list of aspects intercepting control flow structure executions can be found in Appendix B.

```
before(String statement, String clause):
    call(public void Instrumentation.TracingCTRLFlow.ifStatementStart(..))
    && args(statement, clause)

{
  ArrayList log = new ArrayList();

  log.add("If start");
  log.add(statement);
  log.add(clause);
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

```
before():
    call(public void Instrumentation.TracingCTRLFlow.ifStatementEnd())
{
  ArrayList log = new ArrayList();

  log.add("If end");
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

Figure 32 – Aspects intercepting the beginning and end of an if statement

CHAPTER 7 LOGGING AND PARSING TRACES

As mentioned previously, the instrumentation statements will interact with a logging interface. This section attempts to describe the structure of such an interface. Since we are working with a distributed system, each node will be producing trace statements. These traces produced by the instrumented nodes must be stored and combined with other nodes' traces. In order to collect and send the traces to a common server, we propose to have a logging client for each of the instrumented nodes. This logging client runs on the same node the instrumented SUS is running on. It must strive to minimize its impact on the SUS. Each logging client then reports to a logging server (Figure 33). This server contains the logic that joins traces together to generate the instances of the trace metamodel and performs the transformation to create instances of the scenario metamodel. The server is used to centralize the information and perform all the offline processing. The design decisions related to the logging are described in the following sections.

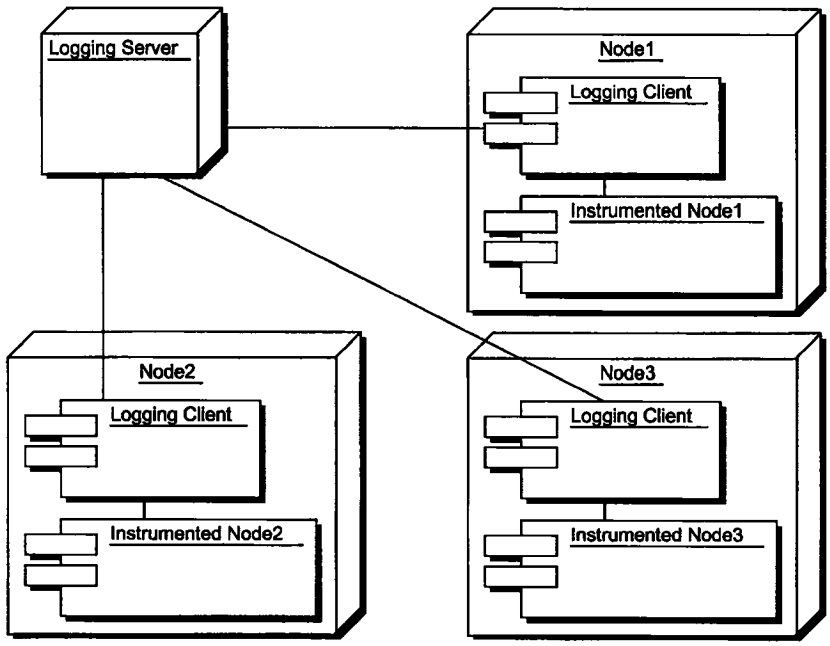


Figure 33 – Logging strategy

7.1 Logging Client

The primary goal of the logging client is to store the trace statements it receives from the instrumented SUS. In addition, it conceals implementation details about the trace storage from the instrumentation aspects and the SUS. The following paragraphs describe how the instrumentation information is sent to the logging client, how it stores this information, how it is first initialized and how it sends the trace to the logging server. The implementation of the logging client is found in Appendix B, Figure 66.

The trace statements are transmitted to the logging client through its `instrument(List, Object[])` method, as mentioned in section 6.4. The first parameter of the method is a `List` of `String` objects that contain the trace statement information. The first `String` in this list is the type of statement. It must be one of the following:

Table 6 – Valid Instrumentation Statement Types

Create method start	Create method end
Static method start	Static method end
Method start	Method end
If start	If end
While start	While end
Do while start	Do while end
For loop start	For loop end
Break	Continue
Start method call	Run method start
Run method end	Remote method call start
Remote method call end	Remote method execution start
Remote method execution end	ThreadComm call start
ThreadComm call end	

The `String` objects that follow in the list are the corresponding information about the statement. For example, in the case where the statement type is "Create method start", the second item in the list contains the name of the class the constructor is being called on and the third the signature of the method.

The second parameter of the `instrument(List, Object[])` method is an array of objects that represent the arguments to the intercepted method. This array may be `null` if there are no arguments. Passing `Objects` makes this method less flexible because the logging client may not be reusable by an instrumented SUS implemented in a programming language other than Java. However, this allows for more customizability in what information is reported. For example, the `collInfo` attribute of `Argument` (as described in section 3.4) contains information about the contents of the collection, if the

argument is of this type. The logging client can be customized to report on a list of the stored elements, the type of elements or their references (to name a few options) when storing the trace statement.

As mentioned previously, once the method is called, additional information is inserted into the trace statement: the thread identifier, the node identifier and the timestamp. As well, delimiters are added between each part of the statement to mark when they each start and stop. This is necessary for parsing, described below. Then, the statement is complete and can be written to the trace file.

The thread identifier is implemented as a hash table of thread objects, as explained in section 6.1.4.2. This table is local to the logging client. The node identifier is also local to the logging client. Its value is obtained from the logging server.

The timestamp is implemented as a simple counter. This is possible because the timestamp is local to each node. Furthermore, the `instrument(List, Object[])` method is synchronized. In other words, only one thread of execution per node can enter the method at a time. This may add an unwanted delay which may change the order of operations. However, it is necessary since any implementation of the timestamp would need to be synchronized. Our approach is simple and avoids a costly system call to get the current time, which may not be precise enough anyhow.

In order to mark the start and end of each piece of information, delimiters are added between statements, between parts of a statement, between arguments and between parts of an argument. Each delimiter must not be a possible character string in the instrumentation statement, as to not confuse the delimiter from the trace information. For

this reason, we cannot use the "endline" character or the "tab" character as a delimiter. Instead, we use a long string of characters (randomly chosen) that is very unlikely to be part of a trace statement. These same separators must be used by the parser (see section 7.2.1).

Once all these elements are added, the trace statement may be written to the trace, a simple text file. Eventually, each trace statement will need to be sorted by thread identifier and by node identifier. To do this simultaneously, each logging client creates a file for each thread and writes a trace statement to the corresponding file. The convention we use for the file name is: TraceThreadXNodeY.txt, where "X" is the thread number and "Y" the node number. Each trace file name is then unique.

Each logging client needs to be initialized before its instrumented node starts to run and generate trace statements. Also, the logging client needs to transmit all trace files once the node is finished running. This is implemented by an aspect that intercepts `public static void main(String[] arg)` methods. Before the main method runs, the logging client contacts the logging server to supply it with a node identifier. After the main method has run, the logging client sends all the trace files it has generated.

7.2 Logging Server

As discussed above, the logging server has the task of sending information, such as the unique node identifier, to the logging clients. It must also collect all the trace files from the clients. However, the logging server's main goal is to perform all of the offline processing and analysis of the traces (Figure 34). This is separated into two main tasks: parsing the traces (section 7.2.1) and performing the transformation (section 7.2.2).

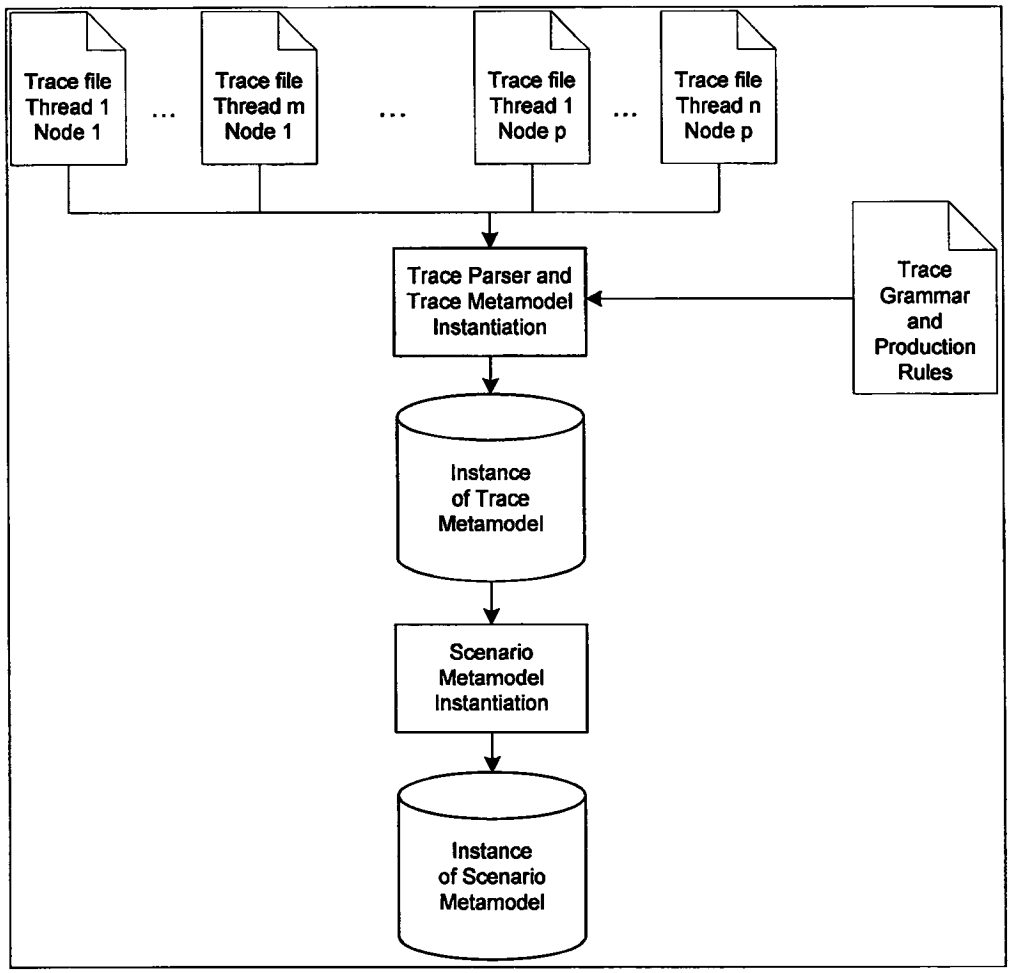


Figure 34 – Processing and Analysis Conducted in Logging Server

7.2.1 Parser

Reading in the files and interpreting the data within is more complex than is appears. Trace statements of different types have various numbers of statement parts that follow it. While the number of parts is predictable from the type of statement, the number of arguments is not: it may vary from zero to many. Below is the BNF grammar that the trace file follows:

```
file := {MethodExecution | RemoteMethodExecution | RunMethodExecution}
MethodExecution := Constructor | StaticMethod | Method | StartThreadCall |
                   RemoteMethodCall | ThreadComm
Constructor := Create method start Integer Integer Integer String String
               {Argument} Body Create method end Integer Integer Integer Integer
StaticMethod := Static method start Integer Integer Integer String String
                String {Argument} Body Static method end Integer Integer Integer
                [Argument]
Method := Method start Integer Integer Integer Integer String String String
          {Argument} Body Method end Integer Integer Integer [Argument]
StartThreadCall := Start method call Integer Integer Integer Integer String
RemoteMethodCall := Remote method call start Integer Integer Integer Remote
                    method call end Integer Integer Integer Integer
ThreadComm := ThreadComm call start Integer Integer Integer Integer String
              String {Argurment} ThreadComm call end Integer Integer Integer
              Integer [Argument]
RemoteMethodExecution := Remote method execution start Integer Integer Integer
                         Integer String Integer Integer String Integer Body
                         Remote method execution end Integer Integer Integer
RunMethodExecution := Run method start Integer Integer Integer Integer String
                      Body Run method end Integer Integer Integer
Body := MethodExecution | IfStatement | Repetition
IfStatement := If start Integer Integer Integer String String Body If end
               Integer Integer Integer
Repetition := While | DoWhile | For
While := While start Integer Integer Integer String String Body While end
         Integer Integer Integer
DoWhile := Do while start Integer Integer Integer Body Do while end Integer
           Integer Integer String String
For := For loop start Integer Integer Integer String String String String Body
       For loop end Integer Integer Integer
Argument := Integer String String String
```

Figure 35 – BNF Grammar of Trace Files

Instead of building our own parser to read in and analyze the trace files, we used the Java Compiler Compiler (JavaCC [17]). This tool takes in the description of the language (Figure 35) and generates the code (in Java) that will parse and analyze the language. The description of the language is similar to BNF productions, which is augmented by Java code, also known as production rules, when certain events (tokens) occur. This description is saved in a file with the extension ".jj", as required by JavaCC.

This file is run through JavaCC, which generates the implementation of the parser (consisting of seven Java classes). The parser reads in the traces and divides the statements into tokens, which are separated by our delimiters. It also verifies that the

number of statement parts is correct. For example, the token "Method end" should always be followed by a timestamp, thread identifier, node identifier and zero or one arguments. An exception is raised if a trace goes against the grammar description.

The Java code we include with the grammar description when certain tokens occur (the production rules) is used to instantiate the appropriate trace metamodel object. For example, when the "Create method start" token is encountered, the code creates an instance of the trace metamodel class `LocalExecution` (with the attribute `isCreate` set to `true`). The other parts of the statement are read in, which specify the timestamp, thread identifier, node identifier, class name, signature and arguments, if any. An instance of the class `Classifier` is then created, as are any `Arguments`. All the corresponding links (as specified in section 3.2.2) are formed.

Note that not all trace statements result in the creation of a trace metamodel class instance. For example, the "for loop end" statement only marks the end of the repetition and that there are no more nested statements within the loop. A "break" and a "continue" statement also do not result in an instance. Instead, these statements replace any number of "if end" statements and the end of one repetition. So that it is not overly complex, these are not present in our grammar described above. Also missing from the grammar is the possibility of encountering the end of a method when the end of an "if" statement or the end of a loop is expected. This is due to a `return` statement in the code. In this case, the end of all "if" statements and the end all loops are implicit. These are handled by the production rules.

7.2.2 Transformation

Once the parser reaches the end of all trace files, the trace metamodel instance is complete. At this point, it needs to be transformed into a scenario metatmodel instance. Because the trace metamodel has many more classes than the scenario metamodel, we need to make them more alike. We accomplish this through various steps: simplification, abstracting remote calls, abstracting starting threads, matching instances of thread communication and finally, creating messages using caller-callee links.

First, we simplify the trace instance by deleting any `Repetition` and `IfStatement` instances that do not have any `MethodExecution` instances within. This is required because ultimately, `MethodExecutions` are transformed into `Messages`. Loops and "if" statements without any `Messages` within do not appear in scenario diagrams. The search for `MethodExecutions` must be recursive because, although there may not be a `MethodExecution` instance in the loop's `nestedStatements`, there may be one in the `nestedStatement` tree structure. If there are no `MethodExecutions`, the `Repetition` or `IfStatement` instance is deleted.

Our next step is to simplify our trace instance by matching the `RemoteCall` instances to the corresponding `RemoteMethodExecution` instance (recall the match description in section 5.1). We eliminate these two objects by linking the `RemoteCall`'s caller to the `RemoteMethodExecution`'s callee with a new `caller-callee` link. This is illustrated in Figure 36. We make a similar link between their nested and nesting statements: the `RemoteCall`'s `nestingStatement` is linked to the `RemoteMethodExecution`'s `nestedStatement` by a `nestingStatement-nestedStatement` association. The

MethodExecution object may be the same as the ExecutionStatement object in the case where the RemoteCall is not within a control flow statement (recall section 4.2). We can then delete all RemoteCall-RemoteMethodExecution pairs.

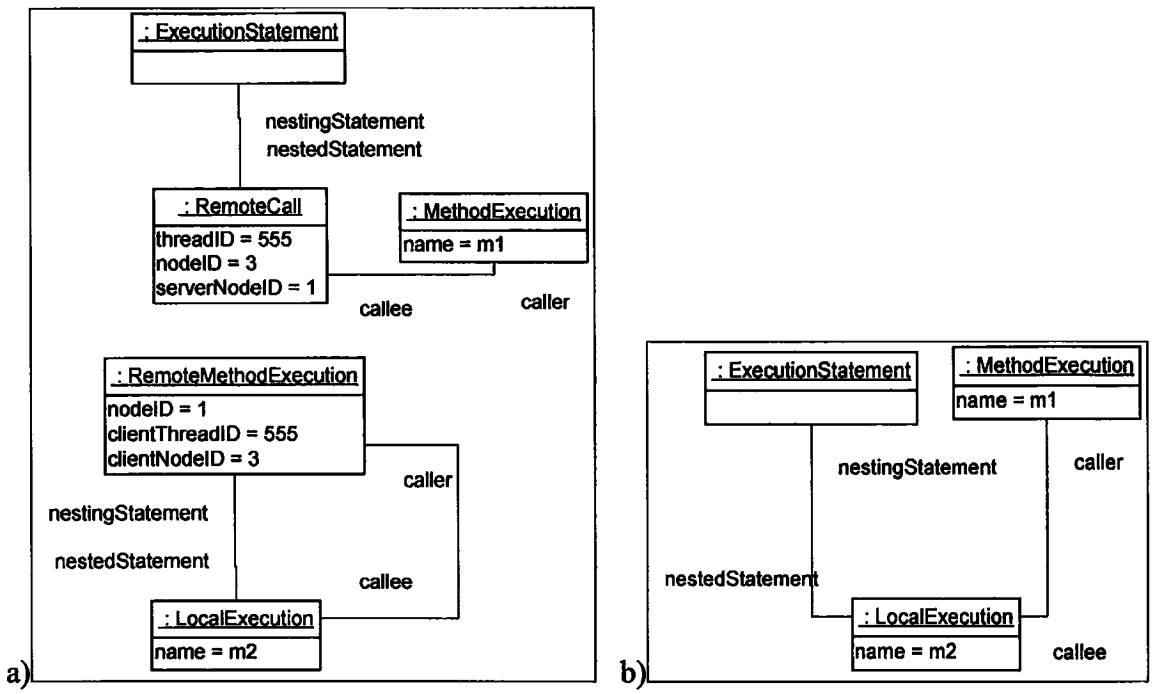


Figure 36 – Transformation by Abstracting Remote Calls and Execution: a) Before Transformation b) After Transformation

The next simplification step repeats a similar process by eliminating StartThreadCall instances. Once the matching LocalExecution instance is found (as described in section 4.4.2), the StartThreadCall instance's caller can be linked with this LocalExecution by a caller-callee link. Again, a similar association is done for the nested and nesting statements: the StartThreadCall instance's nestingStatement can be linked to this same LocalExecution with a nestingStatement-nestedStatement association. The StartThreadCall instance can then be deleted. The LocalExecution is now separated from its former ExecutionStatement. This process is shown in Figure 37.

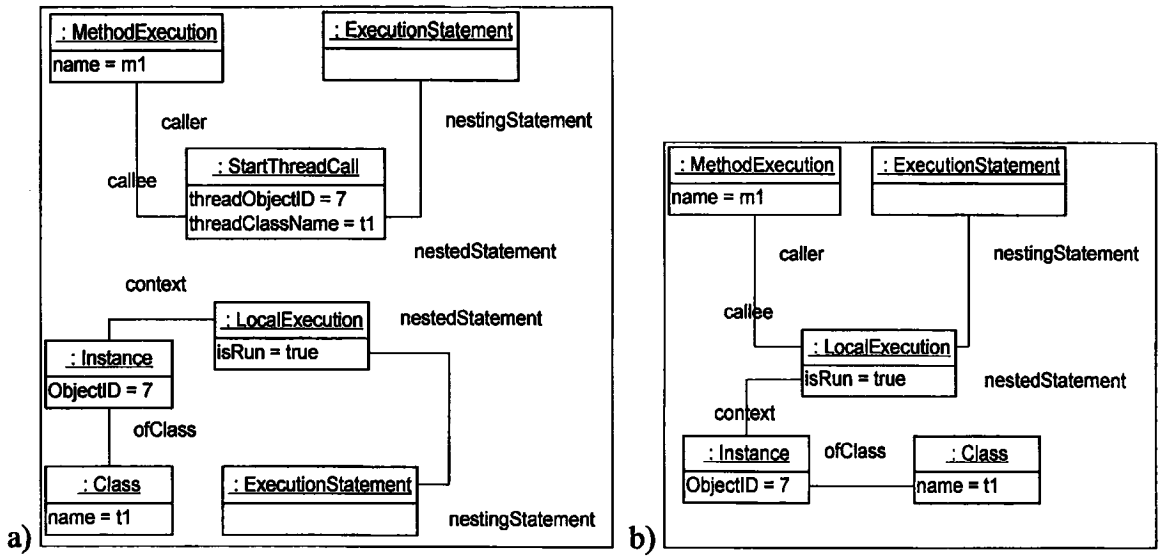


Figure 37 – Transformation by Abstracting Thread Calls:
a) Before Transformation b) After Transformation

The process described above cannot be used for matching `ThreadComm` instances because do not want to delete these types of instances (like we did for `RemoteCalls` and `StartThreadCalls`) because pairs of `ThreadComm` instances will result in a message. As well, they are already part of a caller-callee link that we do not want to replace. Instead, we create a new link called `SignalLink` between `ThreadComm` instances that write and read the same object in the same data structure (see section 5.1). This self-association is temporary, and will help when generating the scenario metamodel instance. It is shown in Figure 38.

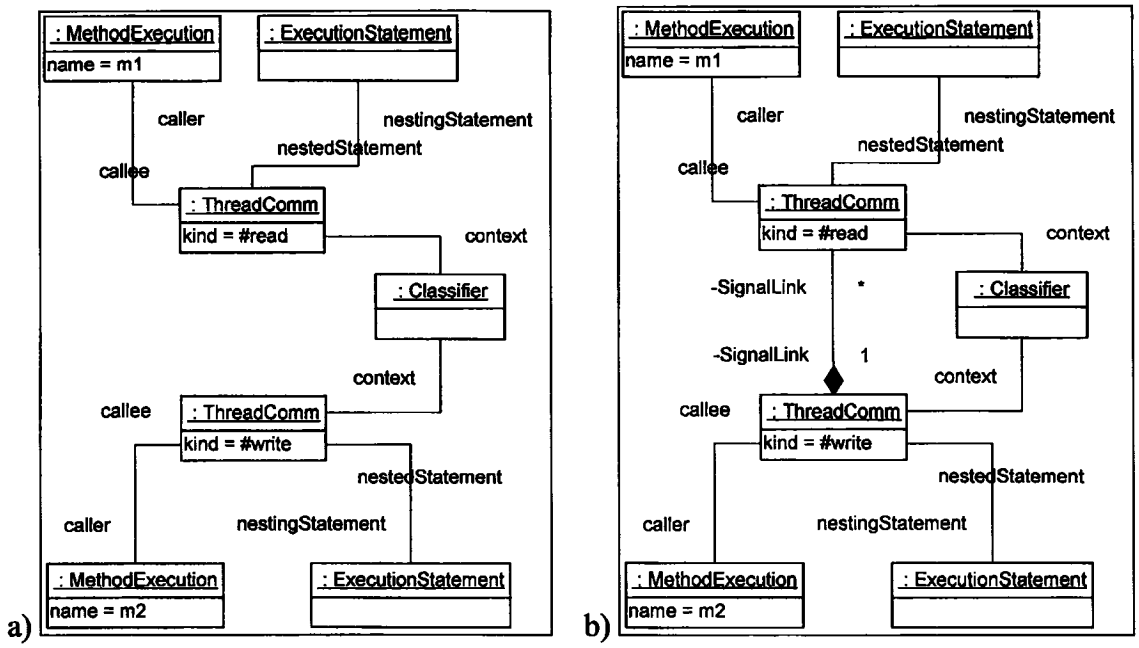


Figure 38 – Adding a Link between matching ThreadComm Objects: a) Without Link b) With Link

Once all these simplifications are complete, the transformation consists of navigating the trace metamodel instance and creating a `Message` instance for each `caller-callee` link. The `Conditions` associated to a `Message` can be found through the operation `obtainConditions()`. The `Repetitions` that the `Message` belongs to can found through the triggers link of each `MethodExecution` (see section 5.3).

CHAPTER 8 CASE STUDY

A Library system was selected 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 in the framework of a fourth year engineering project, by students who were well trained in Analysis and Design using the UML (and OCL) and have good Java programming skills. 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. Note however that the Library system is not multithreaded. Thus, the reverse engineering of asynchronous messages will not be illustrated in this case study, though this was validated on other examples (see Appendix A).

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 Chapter 6. Two exceptions are classes belonging to the Graphical User Interface 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, and they are not described in detail in the

Analysis/Design sequence diagrams: the former is represented by boundary classes [7] and the latter by a Façade [12].

We have selected the "Add copy" use case as a representative example, as it illustrates the most important aspects of the reverse-engineering process. Though this case study may appear of limited size, recall that our focus is on retrieving relevant and complete information on object interactions from dynamic analysis rather than on addressing the visualization problem for large systems.

When the "Add copy" use case is triggered the Analysis/Design documentation indicates that the user must first be authenticated (see sequence diagram in Figure 39). Upon success the use case can proceed: 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 (the barcode is eventually printed on a sticker – message number 12), 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 (message number 7). Now that a new copy has been added for the title, pending reservations, if any (message number 8), on the title can be dealt with: this is the purpose of messages number 9, 10 and 11. Last, the new copy and the title objects can be saved to the database (messages number 13 and 15).

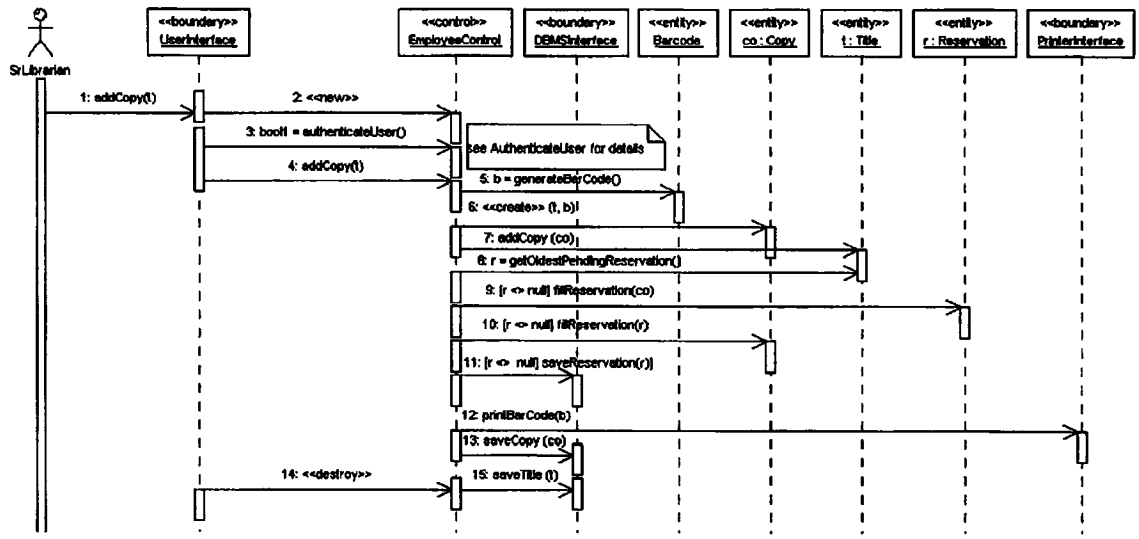


Figure 39 – Sequence diagram provided in Analysis and Design documents

We have executed use case "Add copy" on the instrumented version of the Library. 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 Chapter 5. Figure 40 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 reporting what happens when "Add copy" executes can be found in Appendix C. Figure 40 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` (`objectID=0`) and a remote call (`timestamp=12, threadID=0, nodeID=1, serverNodeID=0`), and the trace at the client side reports on a remote method execution (`clientThreadID=0, clientNodeID=1, className=Employee.EmployeeControlIFacade, clientObjectID=0`).

Client side – nodeID = 1

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

Server Side – nodeID = 0

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

Figure 40 – Excerpt of the trace for use case "Add copy" in Appendix C

Figure 41 shows the scenario diagram[12] corresponding to the trace in Appendix C. This illustrates the usefulness of the approach as discrepancies between this figure (reverse-engineered scenario diagram) and Figure 40 (sequence diagram produced during Design) are clearly visible. However, first note that message number 3 in Figure 39 (user authentication) does not have a counterpart in Figure 41 as, though this has been instrumented, this information has not been included in the trace and the figures we show in this section. Also, not having in Figure 41 counterparts for messages 9 to 11 in Figure 39 is not an error as we have added a copy to a title that does not have reservations (`getOldestPendingReservation()` returns `null`): Figure 41 is only a scenario.

Discrepancies can be found in parameter types: `addCopy()` has a parameter of type `Title` in Figure 39 whereas it has a parameter of type `String` in Figure 41 (the ISBN). Similarly, `Copy`'s constructor has two parameters of types `Title` and `BarCode` in Figure 39 (parameters `t` and `b`) and two parameters of types `String` (the ISBN of the corresponding title) and `int` (the barcode number) in Figure 41. Since the parameter to

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

addCopy() is a String instead of a Title, there is then a need for message getTitle() sent to the database and returning a reference to a Title object.

Figure 41 also indicates that the students decided to store barcode and ISBN numbers instead of Copy and Title references instead of implementing the association between class Title and class Copy. 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, due to their choice of database management system. 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.

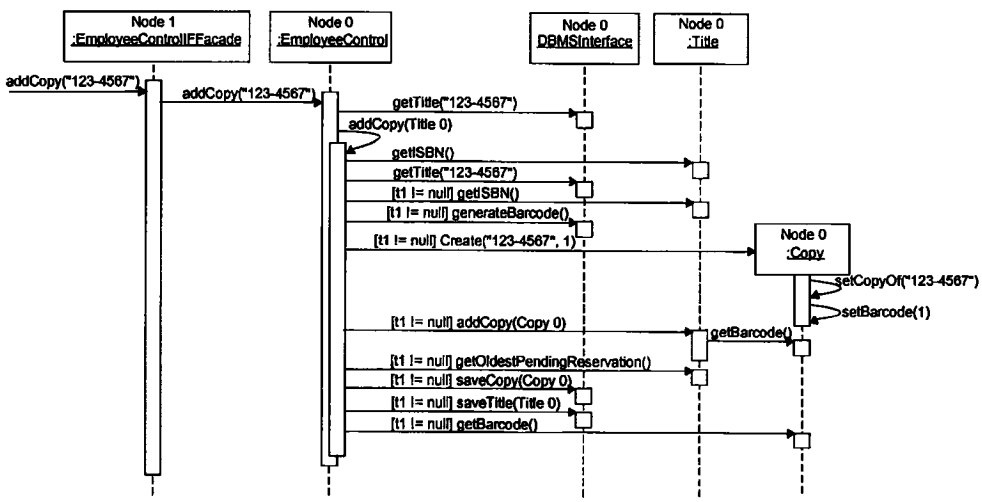


Figure 41 – Scenario diagram produced from the trace in Appendix C

Last, Figure 41 shows un-necessary calls to getISBN() and getBarCode() at the very end of the diagram (after looking at the source code, those calls could have been avoided by using local variables in those two cases). 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 terms of overhead, the execution of "Add copy" went, on average, from 56ms to 86ms when instrumenting the library system, on a PC running Windows XP with a 2.8 MHz processor. Other use cases show similar overheads. Though they are significant, they are not overwhelming: it is not expected that they would prevent the application of our instrumentation approach, except perhaps for hard real-time systems with deadlines.

CHAPTER 9 FUTURE WORK

This section discusses issues that can be addressed when continuing the research presented in this document.

Scenario diagram reverse-engineering is an important step towards program understanding. However, it would be advantageous to continue this work in order to produce sequence diagrams: to abstract information from many scenarios by merging them together to form more general sequences, one for each use-case.

The sequence diagram would have two major differences with our scenario diagrams. First, it would allow branches. Many messages may start at the same point, but with differing guard conditions, and which would show alternative paths of execution. Second, there would no longer be repetition metamodel objects for every time a particular loop is repeated. The executions of a loop would be generalized and need only one repetition instance to represent it.

The sequence diagram metamodel would be very similar to the scenario metamodel, with a few modifications. Two new classes would be added: `Alternative` and `Branch` (Figure 42). A `Branch` represents one possible path of one or more `Messages`. An `Alternative` is a `Message` that has two or more `Branches`. The first `Message` of a `Branch` that is associated to the same `Alternative` must have the same `sourceClassifier` and each must have a `guard`. This is but one possible approach to represent multiple branches of execution.

In its present state, there is not enough information in the trace file to correctly generalize the scenarios. For example, two identical method executions with identical guards and with identical nesting statements may not be from the same point in the code (though it is likely, there is a possibility that it is not: the calls could be made in succession). There are two ways to solve this problem. The first is to combine our approach with static analysis. The trace metamodel would remain the same, as would the scenario metamodel. Information from the code would be needed when instantiating the sequence metamodel. The second solution is to produce trace statements before and after loops and if statements in addition to the trace statements produced presently. Control flow statements would produce trace statements even if they are not executed. This would require modifications to the trace metamodel to accommodate the non-executed control flow statements. Either approach results in obtaining the needed information: the location of "if" statements and repetitions relative to other statements.

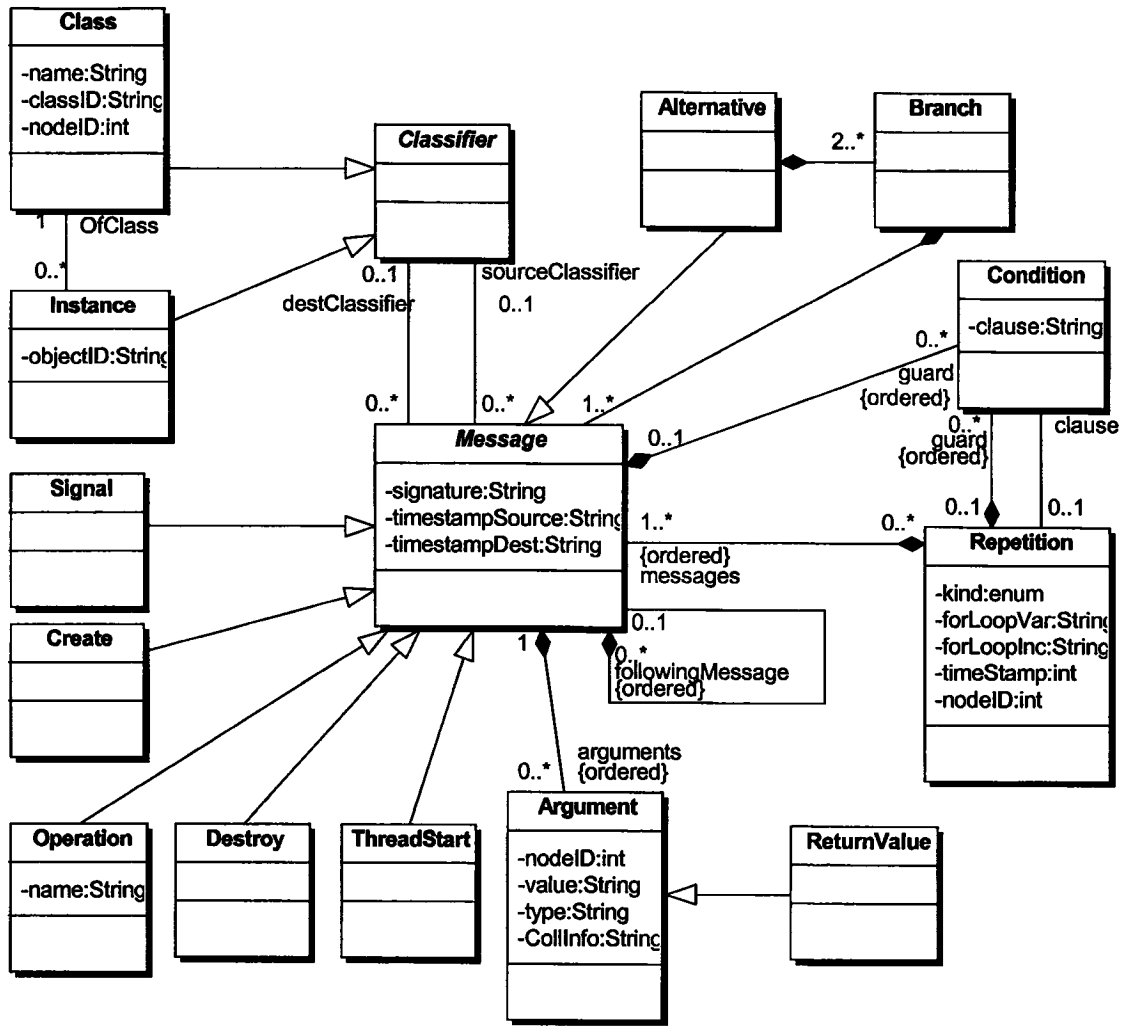


Figure 42 – Possible Class Diagram of Sequence Metamodel

The sequence diagrams generated could then be compared to the original design document diagrams in order to find any discrepancies between the two. One major difficulty is that the reverse-engineered sequence diagram will be much more specific than the design sequence diagram, even after abstracting thread and distributed communication. The reverse-engineered diagram may be correct even though it would contain more messages that the design diagram. It could be possible to analyze the

sequence diagram to see if one is included in the other. That is, compare the diagrams by starting with the first message in the design diagram and finding the analogue message in the generated diagram. Once found, follow the same procedure for the second message, and etc. Reporting on the diagram discrepancies may be complex. An interactive approach may be needed, where the user can accept or reject possible differences. This approach needs to be explored further.

Another improvement worthy of further research is the detection of signals. In this work, signals are limited to communication between threads through a predefined data structure where the object passed must be within a predefined signal hierarchy. Though this is considered good design and implementation practice, this necessitates prior knowledge of the SUS which is undesirable if the tool is being used for studying a system developed by someone other than the user. A different approach to detect signals would be to examine data structures that are referenced by objects of different thread identifiers. This may slow the execution of the testcase due to the examination of all data structures. However, it would be useful to be able to choose which approach will detect signals, as determined by the user's goal, or to be able to interact with the system to identify the data structures and signal objects.

In this work, the implementation assumes an ideal execution of the SUS. Exceptions and lost remote method calls have not been considered. The way these are modeled in the design documents must be evaluated and reflected in the scenario metamodel.

Another improvement involves instrumentation: one major drawback of using AspectJ as an instrumentation strategy is the lack of control flow pointcuts. However, the AspectJ

project is open source. One could determine the feasibility of adding the needed functionality to the implementation by downloading the source code from [2].

Finally, this work does not address diagram visualization. Presently, the scenario metamodel instances are shown as simple text. The instances could be converted to XMI [43], an XML standard used by many tools [5, 26].

CHAPTER 10 CONCLUSION

This thesis provides a comprehensive 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 concurrency and distribution. Though our solution is specific to a Java/RMI context, many of its components can be reused or tailored to other platforms, as further discussed below.

A 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 between metamodels, expressed 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 would be measured according to a global clock instead of local clocks in the case of asynchronous remote communications. Only the instrumentation, discussed next, would then be affected.

In order to minimize the impact of the 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 dealing with concurrency and distribution issues in the context of Java/RMI. If a different middleware were to be used, the aspect code might have to measure time according to a 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.

However, if the instrumentation file has the same format, our tool can still be used. For example, the instrumentation of a system implemented in C++ could result in a trace file of the same structure. Our parser would not make the distinction and create a trace metamodel instance, which would be transformed into a scenario.

We performed a case study on a distributed library management system developed in Java, using RMI as distribution middleware. Our case study 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.

CHAPTER 11 REFERENCES

[1] ANTLR, http://www.antlr.org/

[2] AspectJ, http://eclipse.org/aspectj/

[3] N. Bawa and S. Ghosh, "Visualizing Interactions in Distributed Java Applications," *Proc. IEEE International Workshop on Program Comprehension*, Portland, Oregon, pp. 292-293, May, 2003.

[4] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

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

[6] L. Briand, Y. Labiche, Y. Miao, "Towards the Reverse Engineering of UML Sequence Diagrams", *Proceedings of the IEEE 10th Working Conference on Reverse Engineering (WCRE)*, 2003.

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

[8] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides and J. Yang, "Visualizing the Execution of Java Programs," in S. Diehl, Ed., *Software Visualization*, vol. 2269, *Lecture Notes in Computer Science*, Springer Verlag, 2002, pp. 151-162.

[9] B. P. Douglass, *Real-Time Design Patterns - Robust Scalable Architecture for Real-Time Systems*, Addison-Wesley, 2002.

[10] T. Elrad, R. E. Filman and A. Bader, "Aspect-Oriented Programming: Introduction," in *Communications of the ACM*, vol. 44, 2001, pp. 29-32

[11] H.-E. Eriksson, Magnus Penker, *UML Toolkit*, Wiley Publishing Inc., 1998.

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

[13] J. D. Gradecki, N. Lesiecki, *Mastering AspectJ*, Wiley Publishing Inc., 2003.

[14] R. Hofman and U. Hilgers, "Theory and tool for estimating global time in parallel and distributed systems," *Proc. IEEE Euromicro Workshop on Parallel and Distributed Processing*, Madrid, Spain, pp. 173-179, 1998.

[15] D. Jerding, J. T. Stasko, and T. Ball, "Visualizing Interactions in Program Executions", *Proceeding of the International Conference on Software Engineering*, 1997.

[16] IBM, *Jinsight*, http://www-106.ibm.com/developerworks/library/j-jinsight/

[17] JavaCC, https://javacc.dev.java.net/

[18] R. Kollmann and M. Gogolla, "Capturing Dynamic Program Behaviour with UML Collaboration Diagrams," *Proc. IEEE European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, pp. 58-67, 2001.

[19] 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 Conference on Reverse Engineering*, Richmond, VA, pp. 22-32, 2002.

[20] D. Kortenkamp, R. Simmons, T. Milam and J. L. Fernandez, "A Suite of Tools for Debugging Distributed Autonomous Systems", *Proceedings of the IEEE International Conference on Robotics and Automation*, 2002.

[21] D. B. Lange and Y. Nakamura, "Program Explorer: A Program Visualizer for C++", *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, 1995.

[22] D. Mills, RFC 2030 - Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI, http://www.faqs.org/rfcs/rfc2030.html

[23] J. Moe and D. A. Carr, "Using Execution Trace Data to Improve Distributed Systems", *Software, Practice and Experience*, 2002.

[24] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)," in S. Diehl, Ed., *Software Visualization*, vol. 2269, *Lecture Notes in Computer Science*, Springer Verlag, 2002, pp. 176-190.

[25] OMG, "Unified Modeling Language (UML)," Object Management Group V1.4, www.omg.org/technology/uml/, 2001.

[26] Rational Rose, http://www.rational.com/products/rose/java/jprof.jsp

[27] Rational Test RealTime, http://www.rational.com/products/testrt/index.jsp

[28] M. Raynal and M. Signhal, "Logical Time: A Way to Capture Causality in Distributed Systems," IRISA, Technical Report, January, 1995.

[29] T. Richner and S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles", *Proceeding of International Conference on Software Maintenance*, 2002.

[30] Wm. P. Rogers, *Reveal the Magic Behind Subtype Polymorphism*, Java World, April 2001.

[31] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[32] M. Salah and S. Mancoridis, "Toward an environment for comprehending distributed systems," *Proc. IEEE Working Conference in Reverse Engineering*, Victoria, BC, Canada, pp. 238-247, November, 2003.

[33] W. Schütz, *The Testability of Distributed Real-Time Systems*, Kluwer Academic Publishers, 1993.

[34] S. Si Alhir, *UML in a Nutshell*, O'Reilly & Associates Inc., 1998.

[35] T. Souder, S. Mancoridis and M. Salah, "Form: A Framework for Creating Views of Program Executions", *IEEE Proceedings of the 2001 International Conference on Software Maintenance (ICSM'01)*, 2001.

[36] Sun Microsystems Inc, Java 1.4.2, http://java.sun.com/j2se/1.4.2/docs/api/

[37] Sun, *JVMPI*, http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html

[38] T. Systa, K. Koskimies and H. Muller, "Shimba - An Environment for Reverse Engineering Java Software Systems", *Software - Practice and Experience, vol. 31 (4)*, John Wiley & Sons, 2001.

[39] Y. Terashima, I. Imai, Y. Shimotsuma, Fumiaki Sato and Tadanori Mizuno, "A Proposal of Monitoring and Testing for Distributed Object Oriented Systems", *Electronics and Communications in Japan*, Part 1, Vol. 86, No. 10, 2003.

[40] UML, http://www.omg.org/technology/documents/formal/uml.htm

[41] R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak, "Visualizing dynamic software system information through high-level models", *Proceeding of the 1998 ACM Conference on Object-Oriented Programming, System, Language, and Application (OOPSLA)*, 1998.

[42] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.

[43] XMI, http://www.omg.org/technology/documents/formal/xmi.htm

Appendix A Examples of Trace Metamodel Instances

For each of these examples, we provide a piece of Java source code, a trace execution excerpt (without delimiters) along with the trace metamodel instance, and the corresponding instance of the scenario metamodel.

A.1 A More Complicated Example (example 1)

```
public class A {
  public void m1(String str) {}
  public void m2() {}
  public int m3(B b) {
    m1("arg");
    if(b.attr > 0)
      m2();
    return 5;
  }
}
```

```
public class B {
  public void m4() {
    for(int i = 0; i < 3; i++)
      m5(i);
  }
  public void m5(int x) {attr = x;}
  public int m6() {
    return a.m3(this);
  }
  public B() {
    a = new A();
  }
  public int attr;
  public A a;

  public static void main(String[] arg) {
    B b = new B();
    b.m4();
    b.m6();
  }
}
```

Figure 43 – Source code for example 1

Assuming all the objects have a `threadID` of `0` and a `nodeID` of `0`, the following figure shows the trace produced when executing this example.

```
Static method start 1      0      0      Target.B      public static void
                    Target.B.main(java.lang.String[]) Argument: Node=0
                    Type=[Ljava.lang.String; CollInfo=collection
Constructor start   2      0      0      Target.B      public Target.B()
Constructor start   3      0      0      Target.A      public Target.A()
Constructor end     4      0      0      0
Constructor end     5      0      0      0
Method start        6      0      0      0      Target.B      public void
                    Target.B.m4()
For loop start      7      0      0      for(inti=0;i<3;i++) int i  i<3    i++
Method start        8      0      0      0      Target.B      public void
                    Target.B.m5(int)     Argument: Type=java.lang.Integer Value=0
Method end          9      0      0
For loop end        10     0      0
For loop start      11     0      0      for(inti=0;i<3;i++) int i  i<3    i++
Method start        12     0      0      0      Target.B      public void
                    Target.B.m5(int)     Argument: Type=java.lang.Integer Value=1
Method end          13     0      0
For loop end        14     0      0
For loop start      15     0      0      for(inti=0;i<3;i++) int i  i<3    i++
Method start        16     0      0      0      Target.B      public void
                    Target.B.m5(int)     Argument: Type=java.lang.Integer Value=2
Method end          17     0      0
For loop end        18     0      0
Method end          19     0      0
Method start        20     0      0      0      Target.B      public int
                    Target.B.m6()
Method start        21     0      0      0      Target.A      public int
                    Target.A.m3(Target.B)      Argument: Node=0 Type=Target.B
                    Value=0
Method start        22     0      0      0      Target.A      public void
                    Target.A.m1(java.lang.String)     Argument:
                    Type=java.lang.String Value=arg   test
Method end          23     0      0
If start            24     0      0      if(a.attr > 0a.attr > 0
Method start        25     0      0      0      Target.A      public void
                    Target.A.m2()
Method end          26     0      0
If end              27     0      0
Method end          28     0      0      Return: Type=java.lang.Integer Value=5
Method end          29     0      0      Return: Type=java.lang.Integer Value=5
Static method end   30     0      0
```

Figure 44 – Trace file for example 1

The instance of trace metamodel has been split into two diagrams (Figure 45 and Figure 46). `MethodExecution` instance with `timestamp` 6 at the bottom left of Figure 45 corresponds to `MethodExecution` instance with the same `timestamp` value at the center of Figure 46.

Since only one thread and one node are involved here, attributes `threadID` and `nodeID` are not shown. `Arguments` are not shown either as to not clutter the diagram.

Additionally, links between `Repetition` and `MethodExecution` instances corresponding to association with role name `triggers` have been omitted.

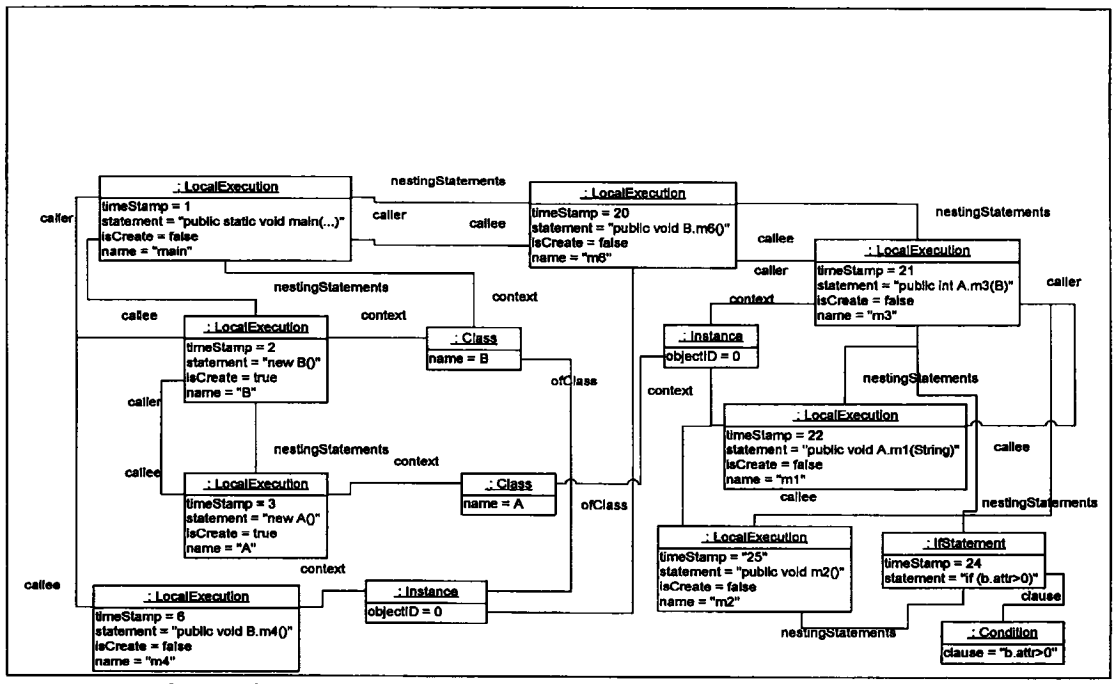


Figure 45 – Trace metamodel instance for example 1 (part I)

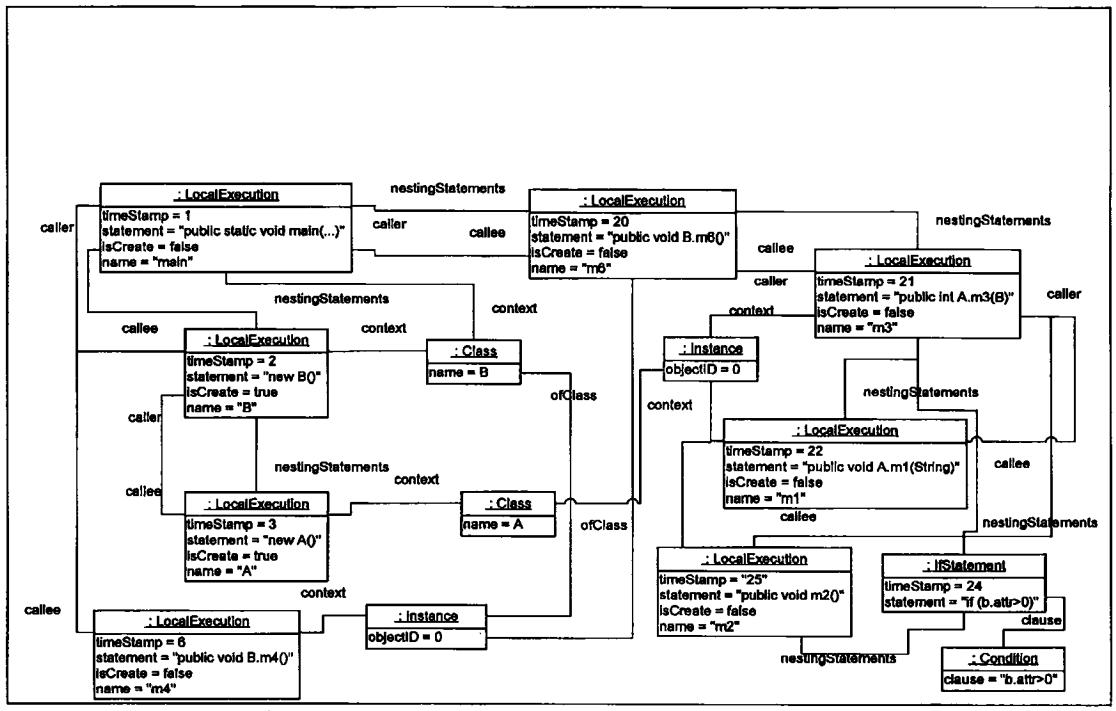


Figure 46 – Trace metamodel instance for example 1 (part II)

The corresponding scenario diagram has also been split into two diagrams (Figure 47 and Figure 48).

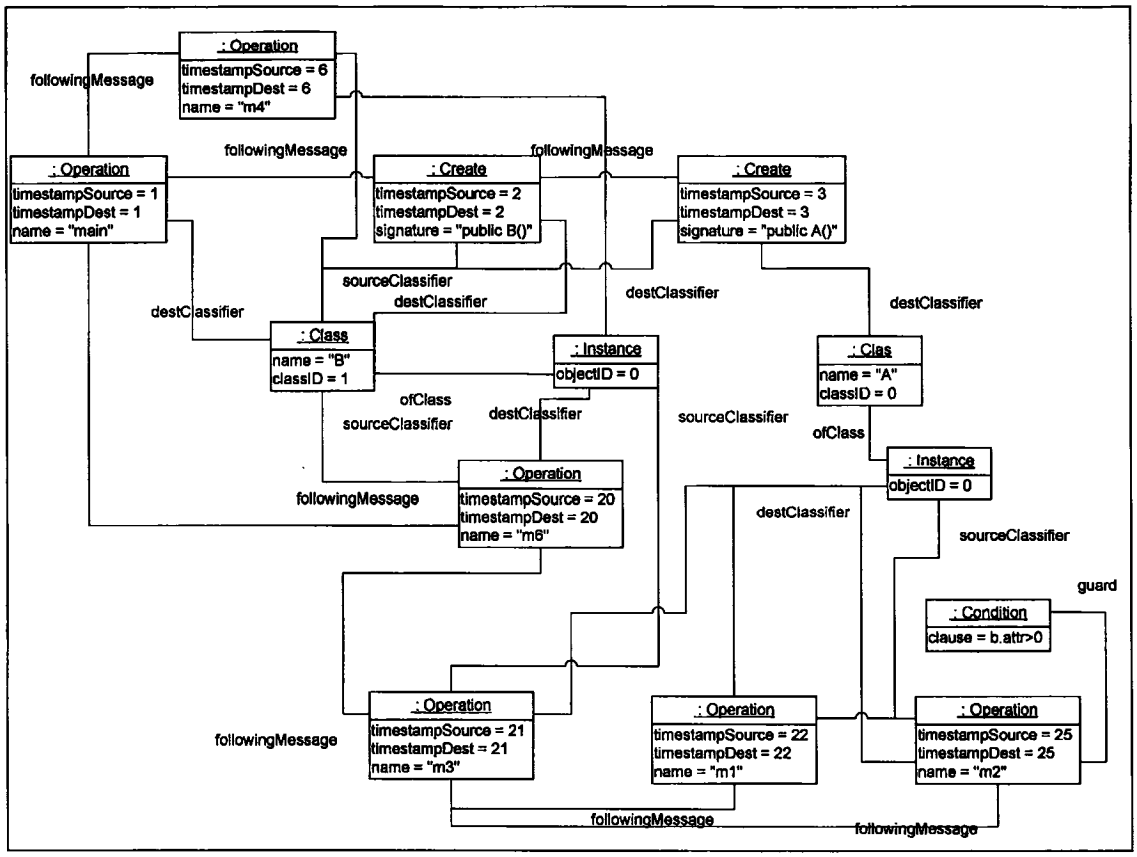


Figure 47 – Scenario metamodel instance for example 1 (part I)

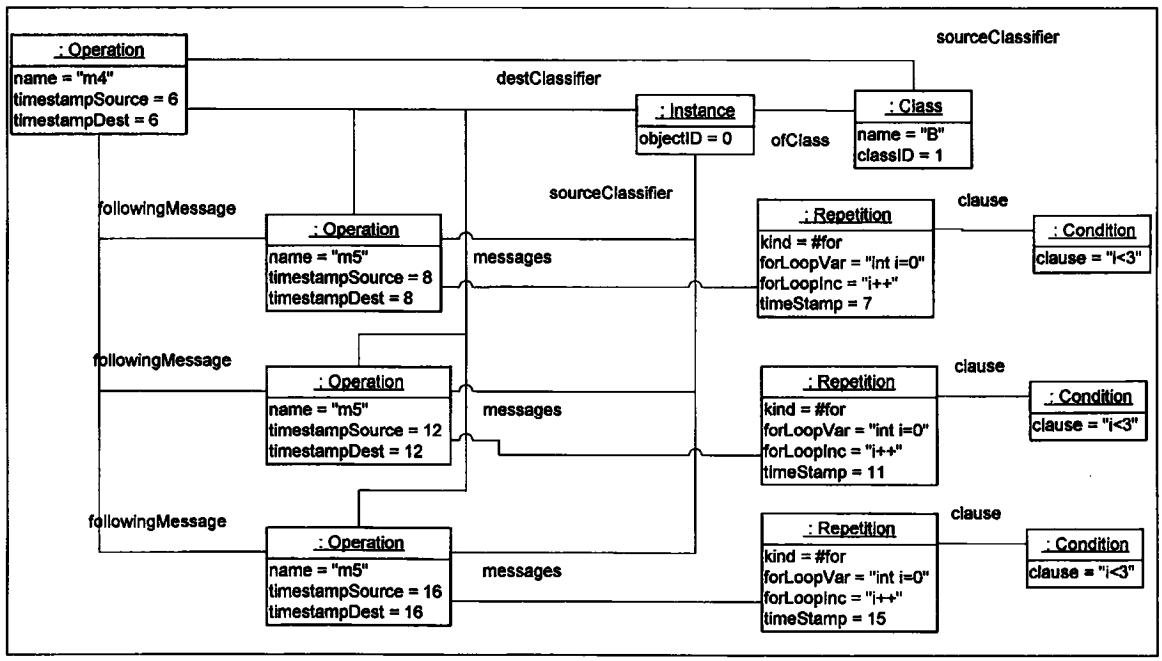


Figure 48 – Scenario metamodel instance for example 1 (part II)

A.2 RMI Communication (example 2)

```
public class A { //nodeID of the client: 0
  public void m1(B server) {
    server.m2("Hello");
    server.m3();}
  public static void main(String[] arg) {
    B b = (B) Naming.lookup("rmi://B_server_address/B_server");
    A client = new A();
    client.m1(b);}
}

public interface B extends Remote {
  public void m2(String str) ;
  public String m3();
}

public class Bimpl extends UnicastRemoteObject implements B {
  //nodeID of the server: 1
  public void m2(String str) { … }
  public String m3() {
    return "Hello to you too";}
  public static void main(String[] arg) {
    Bimpl server = new Bimpl();
    Naming.bind("B_server", server);}
}
```

Figure 49 – Source code for example 2

```
Trace at node 0:
Static method start        1      0      0      Client.A      public static void
                    Client.A.main(java.lang.String[]) Argument: Node=0
                    Type=[Ljava.lang.String; CollInfo=collection
Constructor start          2      0      0      Client.A      public Client.A()
Constructor end            3      0      0      0
Method start                      4      0      0      0      Client.A      public
                    void Client.A.m1(Server.B) Argument: Type=Server.Bimpl_Stub
                    Value=Server.Bimpl_Stub[RemoteStub [ref:
                    [endpoint:[134.117.61.36:3795](remote),objID:[1de3f2d:fca33
                    eb814:-8000, 0]]]]
Remote method call start   5      0      0
Remote method call end     6      0      0      1
Remote method call start   7      0      0
Remote method call end     8      0      0      1
Method end                 9      0      0
Static method end          10     0      0

Trace at node 1: (separated by thread)
Static method start 1      0      1      Server.Bimpl  public static void
                    Server.Bimpl.main(java.lang.String[])   Argument: Node=1
                    Type=[Ljava.lang.String; CollInfo=collection
Constructor start          2      0      1      Server.Bimpl  public
                    Server.Bimpl()
Constructor end            3      0      1      0
Static method end          4      0      1

Remote method execution start     5      1      1      0      0    Client.A      0
Method start               6      1      1      0      Server.Bimpl  public void
                    Server.Bimpl.m2(java.lang.String) Argument:
                    Type=java.lang.String Value=Hello
Method end                 7      1      1
Remote method call end     8      1      1

Remote method execution start     9      1      1      0      0    Client.A      0
Method start               10     1      1      0      Server.Bimpl  public
                    java.lang.String Server.Bimpl.m3()
Method end                        11     1      1      Return:
                    Type=java.lang.String Value=Hello to you too
Remote method call end     12     1      1
```

Figure 50 – Trace file for example 2

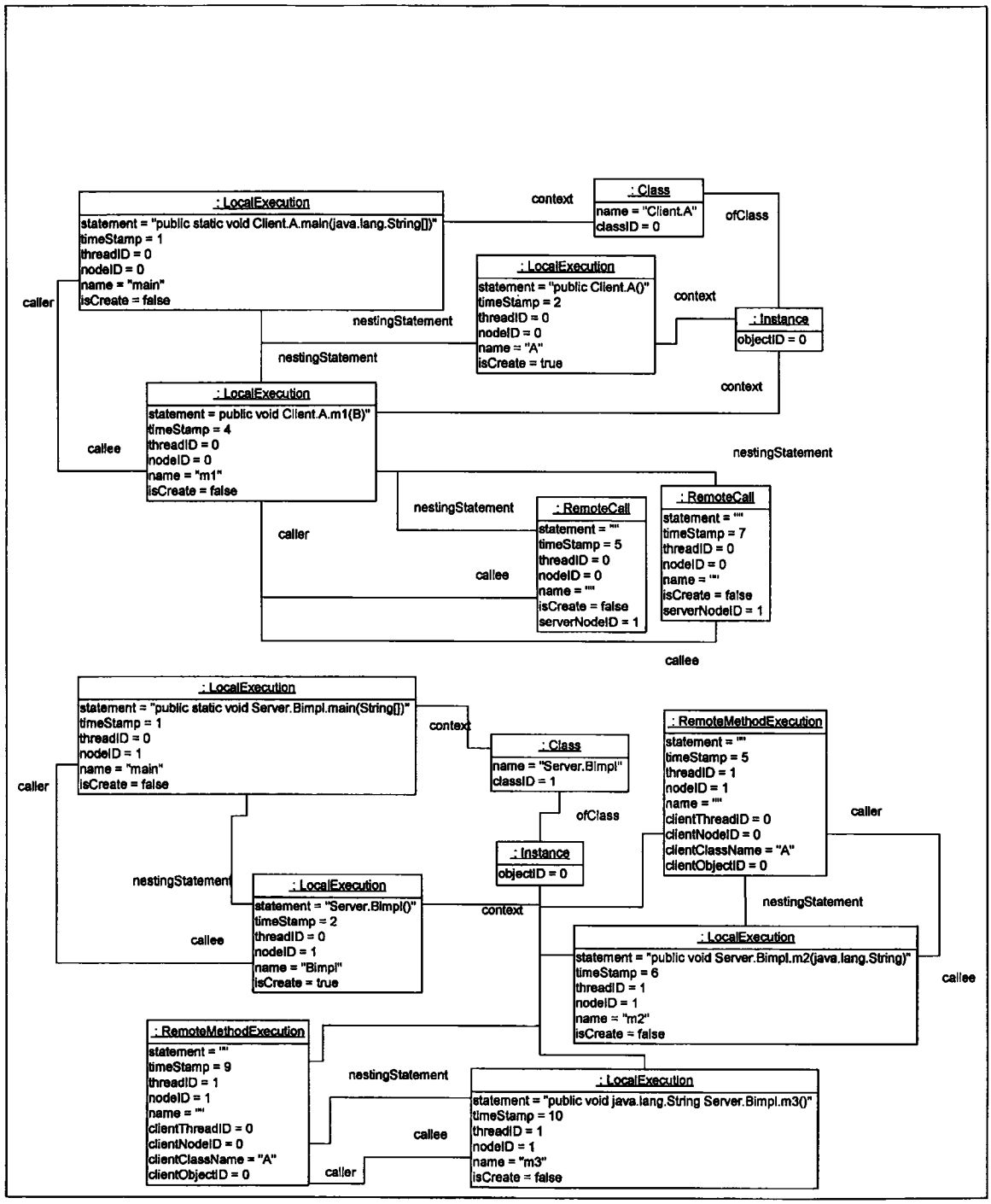


Figure 51 – Trace metamodel instance for example 2

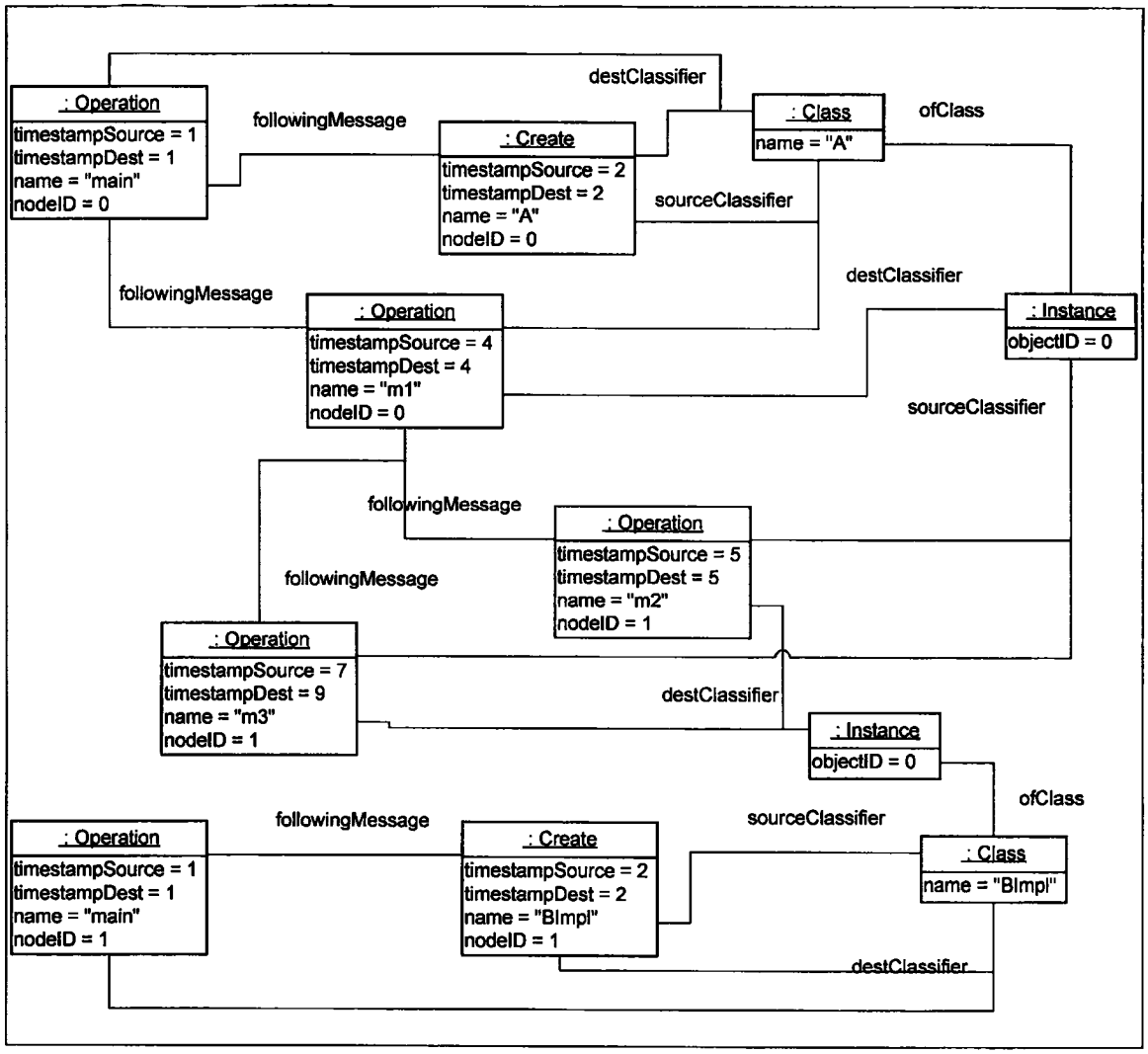


Figure 52 – Scenario diagram metamodel instance for example 2

A.3 Multi-Threading (example 3)

```
public class Producer extends Thread {
private Fifo shared;

public Producer(Fifo shared) {
  this.shared = shared;
}

public void run() {
  int s = (int)(Math.random()*4000);
  for(char ch='A'; ch<='E'; ch++){
    try {
      Thread.sleep(s);
    } catch(InterruptedException e){}
    shared.put(new Character(ch));
    System.out.println(ch +
        " produced by producer.");
  }
}
}
```

```
public class Consumer extends Thread {
private Fifo shared;

public Consumer(Fifo shared) {
  this.shared = shared;
}

public void run() {
  int s = (int)(Math.random()*4000);
  char ch;
  do {
    try {
      Thread.sleep(s);
    } catch(InterruptedException e){}
    ch = ((Character)shared.get()).
              charValue();
    System.out.println(ch +
         " consumed by consumer.");
  } while(ch != 'E');
}
}
```

```
public class ProdCons {
public static void main(String[] arg){
  Fifo shared = new Fifo(2);
  new Producer(shared).start();
  new Consumer(shared).start();
}
}
```

```
public class Fifo {
  private LinkedList fifo;
  private int capacity;
  private int curr_size;

public Fifo(int capacity) {
  this.capacity = capacity;
  fifo = new LinkedList();
  curr_size = 0;
}

synchronized void put(Object obj) {
  if(curr_size >= capacity) {
    try {
    wait();
    } catch (InterruptedException e){}
  }
  fifo.add(obj);
  curr_size++;
  notify();
}

synchronized Object get() {
  if(curr_size == 0) {
    try {
      wait();
    } catch (InterruptedException e){}
  }
  curr_size--;
  notify();
  return fifo.removeFirst();
}
}
```

Figure 53 – Source code for example 3

```
Static method start 1     0     0     Target.ProdCons     public static void
                    Target.ProdCons.main(java.lang.String[])      Argument:
                    Node=0 Type=[Ljava.lang.String; CollInfo=collection
Constructor start   2     0     0     Target.Fifo  public Target.Fifo(int)
                    Argument: Type=java.lang.Integer Value=2
Constructor end     3     0     0     0
Constructor start   4     0     0     Target.Producer     public
                    Target.Producer(Target.Fifo)     Argument: Node=0
                    Type=Target.Fifo Value=0
Constructor end     5     0     0     0
Start method call   6     0     0     0     Target.Producer
Constructor start   8     0     0     Target.Consumer     public
                    Target.Consumer(Target.Fifo)     Argument: Node=0
                    Type=Target.Fifo Value=0
Constructor end     9     0     0     0
Start method call   10    0     0     0     Target.Consumer
Static method end   11    0     0
```

Figure 54 – Trace file for example 3 (for the main thread)

```
Run method start    7     1     0     0     Target.Producer
For loop start      14    1     0     for (char ch = 'A'; ch <= 'E'; ch++)
                    char ch = 'A' ch <= 'E'     ch++
Method start        17    1     0     0     Target.Fifo  synchronized void
                    Target.Fifo.put(java.lang.Object) Argument:
                    Type=java.lang.Character Value=A
Method end          18    1     0
For loop end        19    1     0
For loop start      20    1     0     for (char ch = 'A'; ch <= 'E'; ch++)
                    char ch = 'A' ch <= 'E'     ch++
Method start        25    1     0     0     Target.Fifo  synchronized void
                    Target.Fifo.put(java.lang.Object) Argument:
                    Type=java.lang.Character Value=B
Method end          26    1     0
For loop end        27    1     0
For loop start      28    1     0     for (char ch = 'A'; ch <= 'E'; ch++)
                    char ch = 'A' ch <= 'E'     ch++
Method start        33    1     0     0     Target.Fifo  synchronized void
                    Target.Fifo.put(java.lang.Object) Argument:
                    Type=java.lang.Character Value=C
Method end          34    1     0
For loop end        35    1     0
For loop start      36    1     0     for (char ch = 'A'; ch <= 'E'; ch++)
                    char ch = 'A' ch <= 'E'     ch++
Method start        41    1     0     0     Target.Fifo  synchronized void
                    Target.Fifo.put(java.lang.Object) Argument:
                    Type=java.lang.Character Value=D
Method end          42    1     0
For loop end        43    1     0
For loop start      44    1     0     for (char ch = 'A'; ch <= 'E'; ch++)
                    char ch = 'A' ch <= 'E'     ch++
Method start        45    1     0     0     Target.Fifo  synchronized void
                    Target.Fifo.put(java.lang.Object) Argument:
                    Type=java.lang.Character Value=E
Method end          46    1     0
For loop end        47    1     0
Run method end      48    1     0
```

Figure 55 – Trace file for example 3 (for the producer thread)

```
Run method start    12    2    0    0    Target.Consumer
Do while start      13    2    0
Method start        15    2    0    0    Target.Fifo  synchronized
                    java.lang.Object Target.Fifo.get()
Method end          22    2    0    Return: Type=java.lang.Character
                    Value=A
Do while end        23    2    0    while(ch != 'E')    ch!='E'
Do while start      24    2    0
Method start        29    2    0    0    Target.Fifo  synchronized
                    java.lang.Object Target.Fifo.get()
Method end          30    2    0    Return: Type=java.lang.Character
                    Value=B
Do while end        31    2    0    while(ch != 'E')    ch!='E'
Do while start      32    2    0
Method start        37    2    0    0    Target.Fifo  synchronized
                    java.lang.Object Target.Fifo.get()
Method end          38    2    0    Return: Type=java.lang.Character
                    Value=C
Do while end        39    2    0    while(ch != 'E')    ch!='E'
Do while start      40    2    0
Method start        49    2    0    0    Target.Fifo  synchronized
                    java.lang.Object Target.Fifo.get()
Method end          50    2    0    Return: Type=java.lang.Character
                    Value=D
Do while end        51    2    0    while(ch != 'E')    ch!='E'
Do while start      52    2    0
Method start        53    2    0    0    Target.Fifo  synchronized
                    java.lang.Object Target.Fifo.get()
Method end          54    2    0    Return: Type=java.lang.Character
                    Value=E
Do while end        55    2    0    while(ch != 'E')    ch!='E'
Run method end      56    2    0
```

Figure 56 – Trace file for example 3 (for the consumer thread)

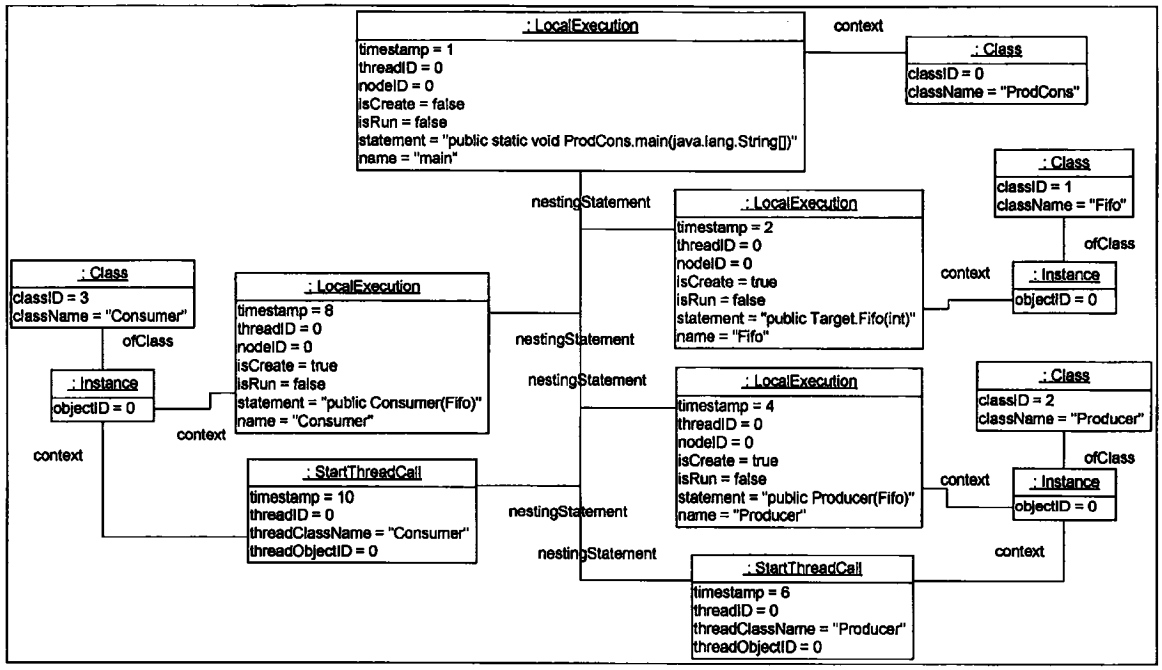


Figure 57 – Trace metamodel instance for example 3 (thread creations)

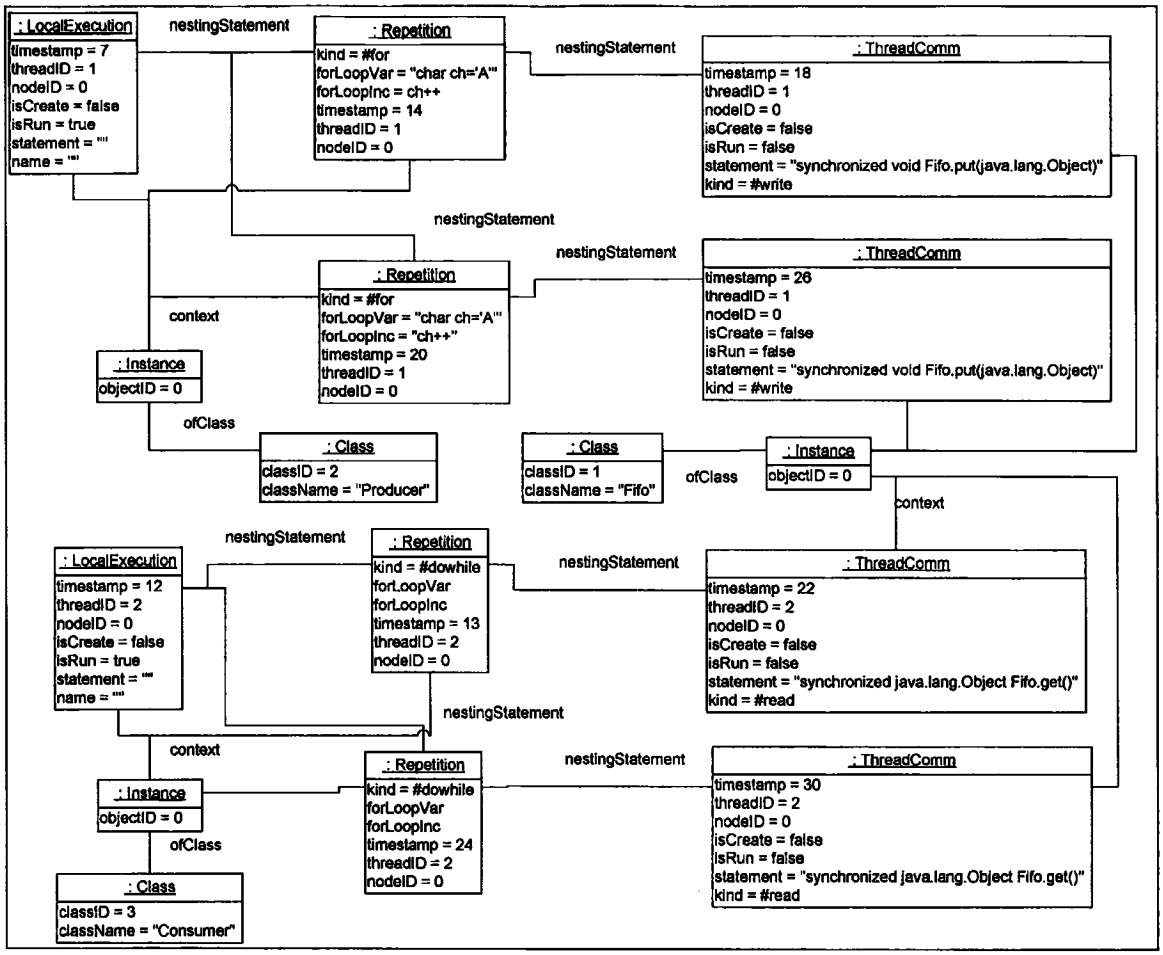


Figure 58 – Trace metamodel instance for example 3 (Producer and Consumer)[13]

[13] Only the two first asynchronous messages are shown (i.e., the first two executions of ThreadComm).

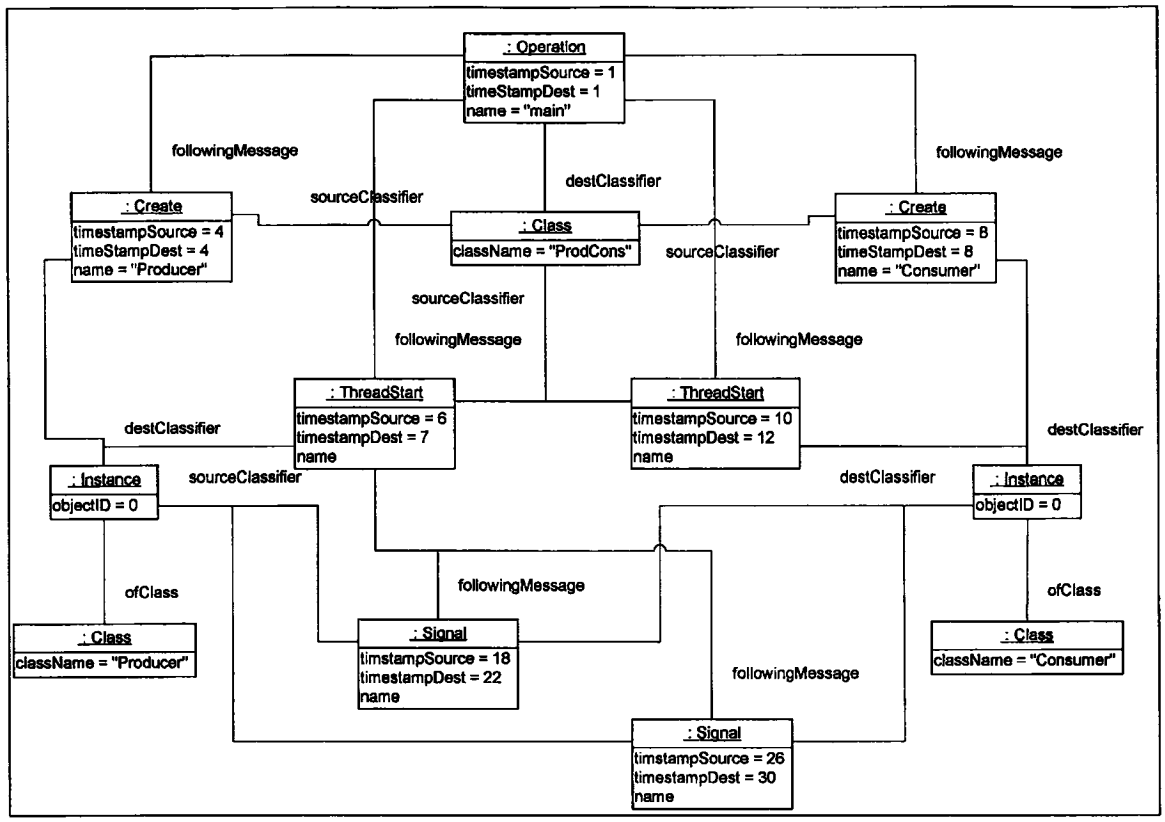


Figure 59 – Scenario diagram metamodel instance for example 3

A.4 Complicated Control Structure

```
public class Loops {
  private static int a;
  public static void m1() {
      a++;
  }

  public static void main(String[] arg) {
      for(int counter = 0; counter < 5; counter++) {
          a = 0;
          while(a < 2) {
              if(counter == 1) {
                  break;
              } else {
                  if(a < 2) {
                      m1();
                  }
              }
          }
          if(counter == 2) {
              return;
          }
      }
  }
}
```

Figure 60 – Source code for example 4

```
Static method start 1      0      0      Target.Loops public static void
                    Target.Loops.main(java.lang.String[])   Argument: Node=0
                    Type=[Ljava.lang.String; CollInfo=collection
For loop start      2      0      0      for (int counter = 0; counter < 5;
                    counter++)    int counter=0 counter<5     counter++
While start         3      0      0      while (a < 2) a<2
If start            4      0      0      else   !(counter==1)
If start            5      0      0      if (a < 2)    a<2
Static method start 6      0      0      Target.Loops public static void
                    Target.Loops.m1()
Static method end   7      0      0
If end              8      0      0
If end              9      0      0
While end           10     0      0
While start         11     0      0      while (a < 2) a<2
If start            12     0      0      else   !(counter==1)
If start            13     0      0      if (a < 2)    a<2
Static method start 14     0      0      Target.Loops public static void
                    Target.Loops.m1()
Static method end   15     0      0
If end              16     0      0
If end              17     0      0
While end           18     0      0
For loop end        19     0      0
For loop start      20     0      0      for (int counter = 0; counter < 5;
                    counter++)    int counter=0 counter<5     counter++
While start         21     0      0      while (a < 2) a<2
If start            22     0      0      if (counter == 1)   counter==1
Break or Continue   23     0      0
For loop end        24     0      0
For loop start      25     0      0      for (int counter = 0; counter < 5;
                    counter++)    int counter=0 counter<5     counter++
While start         26     0      0      while (a < 2) a<2
If start            27     0      0      else   !(counter==1)
If start            28     0      0      if (a < 2)    a<2
Static method start 29     0      0      Target.Loops public static void
                    Target.Loops.m1()
Static method end   30     0      0
If end              31     0      0
If end              32     0      0
While end           33     0      0
While start         34     0      0      while (a < 2) a<2
If start            35     0      0      else   !(counter==1)
If start            36     0      0      if (a < 2)    a<2
Static method start 37     0      0      Target.Loops public static void
                    Target.Loops.m1()
Static method end   38     0      0
If end              39     0      0
If end              40     0      0
While end           41     0      0
If start            42     0      0      if (counter == 2)   counter==2
Static method end   43     0      0
```

Figure 61 – Trace file for example 4

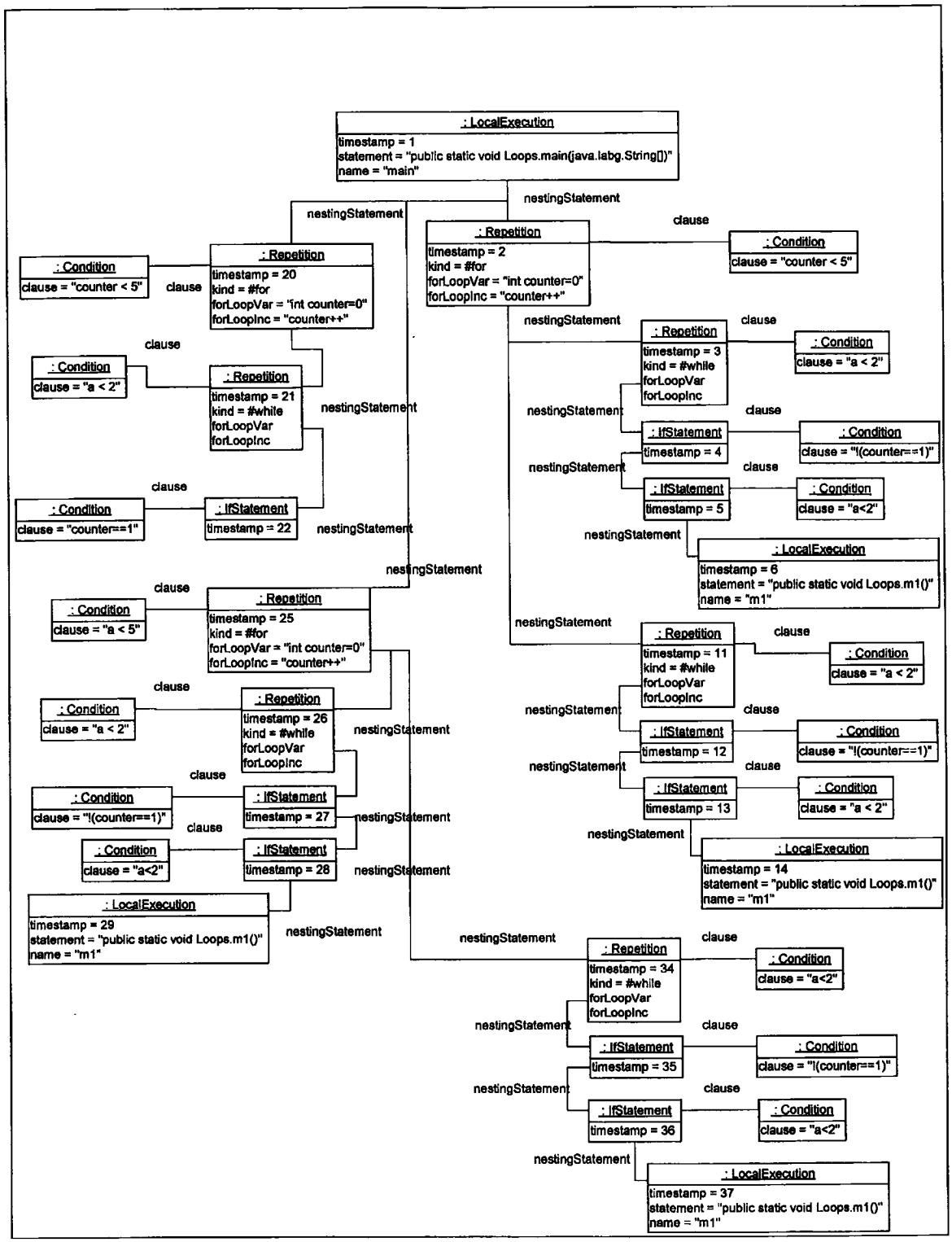


Figure 62 – Trace metamodel instance for example 4

In the trace metamodel instance, the (unique) context (i.e., class Loops) as well as `caller-callee` links are not shown to not clutter the diagram. Similarly, `nodeIDs` and `threadIDs` are not shown.

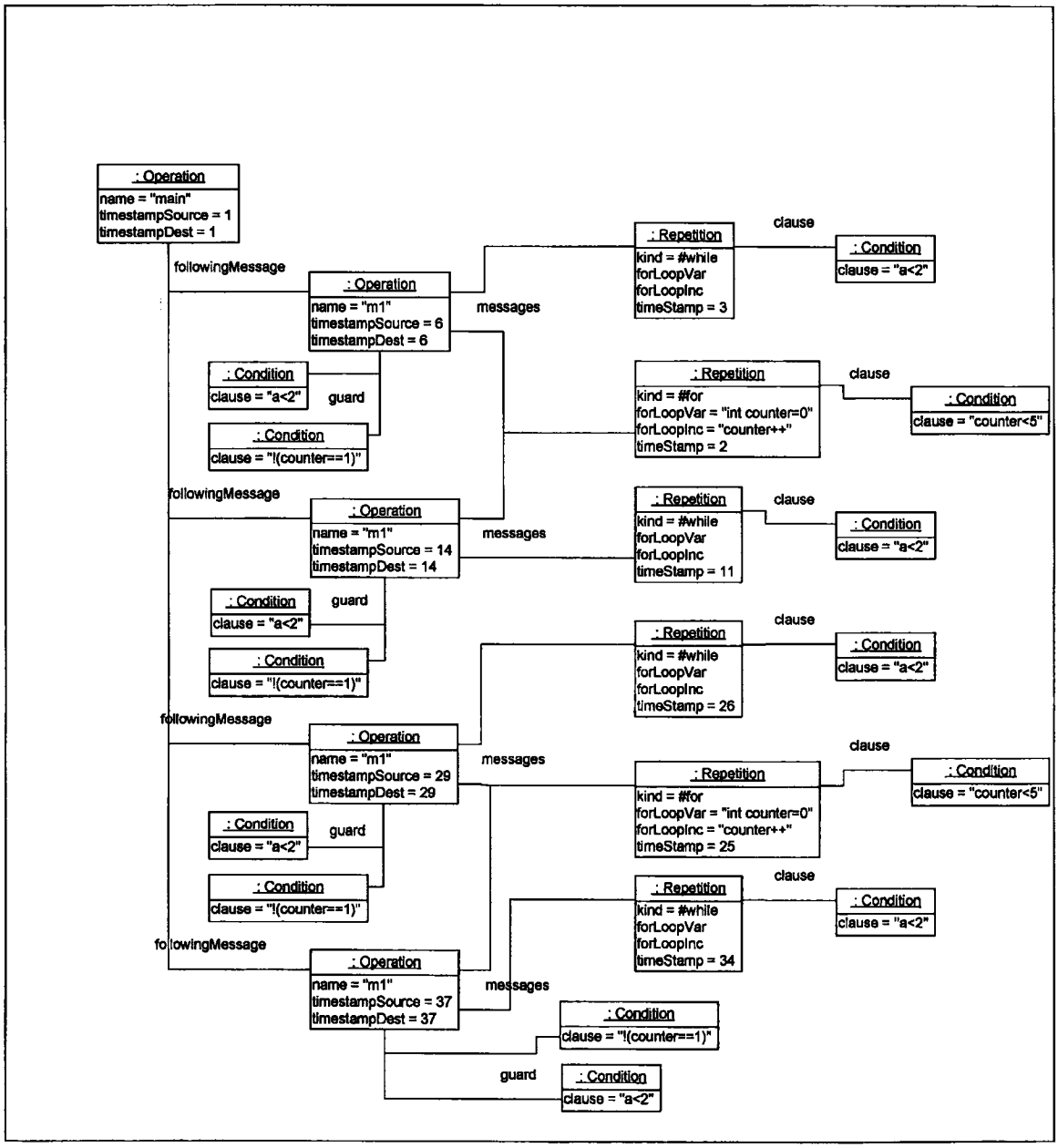


Figure 63 – Scenario diagram metamodel instance for example 4

Appendix B Complete List of AspectJ Templates

B.1 Utility classes within the aspects

```
public class CollIDmap {
  private static CollIDmap instance = new CollIDmap();
  private CollIDmap() { }
  public static CollIDmap getCollIDmap() { return instance; }
  private HashMap hash = new HashMap();
  private int currentCollID = 0;
  public synchronized int getCollID(Object o) {
    if(hash.containsKey(o))
      return ((Integer)hash.get(o)).intValue();
    else {
      hash.put(o, new Integer(currentCollID));
      return currentCollID++;
    }
  }
}
```

Figure 64 – Class `CollIDmap`

```
public class TracingCTRLFlow {
  public static void ifStatmentStart( String Statement,
                                      String clause) {}
  public static void ifStatmentEnd() {}

  public static void whileStatmentStart( String Statement,
                                         String clause) {}
  public static void whileStatmentEnd() {}

  public static void doWhileStatmentStart() {}
  public static void doWhileStatmentEnd( String Statement,
                                         String clause) {}

  public static void forStatmentStart( String Statement,
                                       String var,
                                       String clause,
                                       String inc) {}
  public static void forStatmentEnd() {}

  public static void breakStatment() {}
  public static void continueStatment() {}
}
```

Figure 65 – Class `TracingCTRLFlow`

```
public class LoggingClient
{
   public static LoggingClient instance = new LoggingClient();
   private LoggingClient() { }
   private ArrayList fileWriterList = new ArrayList();
   private HashMap threadIDhash = new HashMap();
   private int currentThreadID = 0;
   private int currentTimestep = 1;
   private int nodeID = 0;
   private final String DELIM_IN_REC = " OBSCURE_DELIMITER ";
   private final String DELIM_BETW_REC = " OBSCURE_DELIMITER_ENDLINE ";
   private final String DELIM_IN_ARG = " OBSCURE_DELIMITER_IN_ARG ";
   private final String DELIM_BETW_ARG = " OBSCURE_DELIMITER_BETWEEN_ARGS ";

   public static LoggingClient getLoggingClient() {return instance; }

   public void setNodeID(int node) {nodeID = node; }
   public int getNodeID() {return nodeID; }

   public UniqueID getUniqueID(String cn) {
       try {
          return new UniqueID(cn, nodeID, getThreadID());
       } catch (IOException e) {
          System.out.println("Exception in Logging Client: " + e);
       }
       return null;
   }

   public UniqueID getUniqueID(String cn, int obj) {
       try {
          return new UniqueID(cn, obj, nodeID, getThreadID());
       } catch (IOException e) {
          System.out.println("Exception in Logging Client: " + e);
       }
       return null;
   }

   public void instrument(List record, Object[] arguments) {
       try {
          int thread = getThreadID();
          ((FileWriter) fileWriterList.get(thread)).write((String)
record.get(0));
          ((FileWriter) fileWriterList.get(thread)).write(DELIM_IN_REC);
          ((FileWriter)
fileWriterList.get(thread)).write(String.valueOf(currentTimestep));
          currentTimestep++;
          ((FileWriter) fileWriterList.get(thread)).write(DELIM_IN_REC);
          ((FileWriter) fileWriterList.get(thread)).
                                        write(String.valueOf(thread));
          ((FileWriter) fileWriterList.get(thread)).write(DELIM_IN_REC);
          ((FileWriter) fileWriterList.get(thread)).
                                        write(String.valueOf(nodeID));
          ((FileWriter) fileWriterList.get(thread)).write(DELIM_IN_REC);

          for (int count = 1; count < record.size(); count++) {
              ((FileWriter) fileWriterList.get(thread)).write((String)
record.get(count));
              ((FileWriter) fileWriterList.get(thread)).write(DELIM_IN_REC);
          }
          if (arguments != null)
              ((FileWriter) fileWriterList.get(thread)).
                                             write(argsToString(arguments));
          ((FileWriter) fileWriterList.get(thread)).write(DELIM_BETW_REC);
```

```
            ((FileWriter) fileWriterList.get(thread)).flush();
        } catch (IOException e) {
            System.out.println("Error in Logging Client: " + e);
        }
    }

    private int getThreadID() throws IOException {
        Thread thread = Thread.currentThread();
        if (threadIDhash.containsKey(thread))
            return ((Integer) threadIDhash.get(thread)).intValue();
        else {
            threadIDhash.put(thread, new Integer(currentThreadID));
            fileWriterList.add(currentThreadID, new FileWriter(new
File("TraceNode" + nodeID + "Thread" + currentThreadID + ".txt")));
            return currentThreadID++;
        }
    }

    public String argsToString(Object[] arg) {
        StringBuffer str = new StringBuffer();
        for (int i = 0; i < arg.length; i++) {
            if(arg[i] != null) {
                str.append(argsToString(arg[i]));
                if (i < arg.length - 1)
                str.delete(str.length()-DELIM_BETW_ARG.length()-1,
                           str.length()-1); //delimiter
            }
        }
        return str.toString();
    }

    public String argsToString(Object arg) {
        StringBuffer str = new StringBuffer();
        str.append(DELIM_BETW_ARG); //delimiter
        if (arg instanceof ObjectID) {
            str.append(nodeID);
            str.append(DELIM_IN_ARG); //delimiter
            str.append(arg.getClass().getName());
            str.append(DELIM_IN_ARG); //delimiter
            str.append(((ObjectID) arg).getObjectID());
            str.append(DELIM_IN_ARG); //delimiter
        } else if ((arg instanceof Collection) || (arg instanceof Map)
                           || (arg.getClass().isArray()))
        {
            str.append(nodeID);
            str.append(DELIM_IN_ARG); //delimiter
            str.append(arg.getClass().getName());
            str.append(DELIM_IN_ARG); //delimiter
            str.append(DELIM_IN_ARG); //delimiter
            str.append("collection"); //optional
        } else {
            str.append(DELIM_IN_ARG); //delimiter
            str.append(arg.getClass().getName());
            str.append(DELIM_IN_ARG); //delimiter
            str.append(arg.toString());
            str.append(DELIM_IN_ARG); //delimiter
        }
        str.append(DELIM_BETW_ARG);
        return str.toString();
    }
}
```

Figure 66 – Class `LoggingClient`

B.2 Additional aspect templates

```
public Object  InterfaceName.MethodNameExtra(
                        UniqueID client
                        [, parameters of the method - if any]
               ) throws RemoteException [, other throws clauses - if any]
{
  ArrayList log = new ArrayList();

  log.add("Remote method execution start");
  log.add(String.valueOf(((ObjectID) this).getObjectID()));
  log.add(this.getClass().getName());
  log.add(String.valueOf(client.threadID));
  log.add(String.valueOf(client.nodeID));
  log.add(client.className);
  log.add(String.valueOf(client.objectID));
  LoggingClient.getLoggingClient().instrument(log, null);

  MethodName([arguments of the method - if any]);

  ArrayList retArray = new ArrayList(2);
  retArray.add(0, "dummy");
  retArray.add(1, new Integer(LoggingClient.getLoggingClient().getNodeID()));

  log.clear();
  log.add("Remote method execution end");
  LoggingClient.getLoggingClient().instrument(log, null);

  return retArray;
}
```

Figure 67 – Aspect template for tracing execution of remote methods without any return value (recall Figure 26 in Section 6.4.3.1)

```
before(String statement, String clause):
    call(public void Instrumentation.TracingCTRLFlow.whileStatementStart(..))
    && args(statement, clause)
{
    ArrayList log = new ArrayList();

     log.add("While start");
     log.add(statement);
     log.add(clause);
     LoggingClient.getLoggingClient().instrument(log, null);
}
before():
    call(public void Instrumentation.TracingCTRLFlow.whileStatementEnd(..))
{
    ArrayList log = new ArrayList();

    log.add("While end");
    LoggingClient.getLoggingClient().instrument(log, null);
}
```

Figure 68 – Aspect for intercepting `while` loops

```
before():
    call(public void Instrumentation.TracingCTRLFlow.doWhileStatementStart(..))
{
  ArrayList log = new ArrayList();

  log.add("Do while start");
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

```
before(String statement, String clause):
    call(public void Instrumentation.TracingCTRLFlow.doWhileStatementEnd(..))
    && args(statement, clause)
{
  ArrayList log = new ArrayList();

  log.add("Do while end");
  log.add(statement);
  log.add(clause);
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

Figure 69 – Aspect for intercepting `do-while` loops

```
before(String statement, String var, String clause, String inc):
    call(public void Instrumentation.TracingCTRLFlow.forStatementStart(..))
    && args(statement, var, clause, inc)
{
  ArrayList log = new ArrayList();

  log.add("For loop start");
  log.add(statement);
  log.add(var);
  log.add(clause);
  log.add(inc);
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

```
before():
    call(public void Instrumentation.TracingCTRLFlow.forStatementEnd(..))
{
  ArrayList log = new ArrayList();

  log.add("For loop end");
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

Figure 70 – Aspect for intercepting `for` loops

```
before():
    call(public void Instrumentation.TracingCTRLFlow.breakStatement(..))
{
  ArrayList log = new ArrayList();

  log.add("Break");
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

```
before():
    call(public void Instrumentation.TracingCTRLFlow.continueStatement(..))
{
  ArrayList log = new ArrayList();

  log.add("Continue");
  LoggingClient.getLoggingClient().instrument(log, null);
}
```

Figure 71 – Aspect for intercepting `breaks` and `continues`

Appendix C Example Trace for Case Study

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

Figure 72 – Example of trace for the Library system (client side)

```
Remote method execution start     55     1      0      server.EmployeeControl
                    0      1      Employee.EmployeeControlIFFacade 0
Method start 56     1      0      0      server.EmployeeControl      public long
             server.EmployeeControl.addCopy(java.lang.String)      addCopy
             java.lang.String     123-4567
Static method start 57     1      0      server.DBMSInterface        public static
             server.Title server.DBMSInterface.getTitle(java.lang.String)
                    getTitle      java.lang.String     123-4567
Static method end   58     1      0      0      server.Title 0
Method start 59     1      0      0      server.EmployeeControl      public long
             server.EmployeeControl.addCopy(server.Title)   addCopy       0
             server.Title 0
Method start 60     1      0      0      server.Title public java.lang.String
             server.Title.getISBN()     getISBN
Method end   61     1      0      java.lang.String     123-4567
Static method start 62     1      0      server.DBMSInterface        public static
             server.Title server.DBMSInterface.getTitle(java.lang.String)
             getTitle      java.lang.String     123-4567
Static method end   63     1      0      0      server.Title 0
If start     64     1      0      if (t1 != null)      t1!=null
Method start 65     1      0      0      server.Title public java.lang.String
             server.Title.getISBN()     getISBN
Method end   66     1      0      java.lang.String     123-4567
Static method start 67     1      0      server.DBMSInterface        public static
             long server.DBMSInterface.generateBarcode()    generateBarcode
Static method end   76     1      0      java.lang.Long       1
Constructor start   77     1      0      server.Copy  public
             server.Copy(java.lang.String, long)      java.lang.String    123-
             4567   java.lang.Long       1
Method start 78     1      0      0      server.Copy  public void
             server.Copy.setCopyOf(java.lang.String)        setCopyOf
                    java.lang.String     123-4567?
Method end   79     1      0
Method start 80     1      0      0      server.Copy  public void
             server.Copy.setBarCode(long)      setBarCode    java.lang.Long    1
Method end   81     1      0
Constructor end     82     1      0      0
Method start 83     1      0      0      server.Title public void
             server.Title.addCopy(server.Copy)        addCopy       0
             server.Copy  0
Method start 84     1      0      0      server.Copy  public long
             server.Copy.getBarCode()   getBarCode
Method end   85     1      0      java.lang.Long       1
Method end   86     1      0
Method start 87     1      0      0      server.Title public server.Reservation
             server.Title.getOldestPendingReservation()
             getOldestPendingReservation
Method end   88     1      0
Static method start 89     1      0      server.DBMSInterface        public static
             void server.DBMSInterface.saveCopy(server.Copy)        saveCopy
             0      server.Copy  0
Static method end   104    1      0
Static method start 105    1      0      server.DBMSInterface        public static
             void server.DBMSInterface.saveTitle(server.Title)      saveTitle
             0      server.Title 0
Static method end   130    1      0
Method start 131    1      0      0      server.Copy  public long
             server.Copy.getBarCode()   getBarCode
Method end   132    1      0      java.lang.Long       1
If end       133    1      0
Method end   134    1      0      java.lang.Long       1
Method end   135    1      0      java.lang.Long       1
Remote method execution end       136    1      0
```

Figure 73 – Example of trace for the Library system (server side)