Chair of
Software Engineering

Capture and Replay for Eiffel

Stefan Sieber

02-911-790
Master Thesis
Summer 2007

Chair of Software Engineering
Department of Computer Science
ETH Zürich

Andreas Leitner
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

Debugging applications is very tedious work. One of the hardest steps to find a fault in a program is to reproduce the steps that lead to the associated program failure. Capture and replay aims at making the execution of a program repeatable, removing this burden from the developer. In order to be able to replay an application, it is necessary to isolate the deterministic part of the application that needs to be replayed (observed part) from the non-deterministic parts, like user input, storage or network (unobserved part). Once the behaviour of the unobserved part can be captured and replayed, it is also possible to replay the observed part. Although capture and replay is commonly known as capturing mouse and keyboard events, there are approaches that go further and capture events from all interactions with the environment. All these approaches have one thing in common: They have a fixed border between observed and unobserved part. Selective capture and replay [13] makes it possible to individually define the observed part, thus offers the possibility to optimize the amount of information that needs to be captured. Another unique feature of selective capture and replay is that the amount of recorded data is linear to the number of variables passed over the border between the observed and unobserved part, whereas conventional systems record all the data that passes the border.

The goal of this thesis is to implement selective capture replay for Eiffel, which in contrast to Java, supports multiple inheritance. This makes it necessary to instrument the programs in a different way than in the original implementation. This changed implementation offers the possibility to switch between capture and replay phase without recompilation. To validate the implementation, an example will be presented and the performance of the technique will be measured using this example.

Acknowledgements

"If I have seen a little further it is by standing on the shoulders of giants."

Sir Isaac Newton

This thesis can never be compared to the works of Newton, but it has one thing in common with them: It would never have been possible without the achievements and support of others. Here, I'd like to thank all the 'giants' that have let me stand on their shoulders the past six months:

First of all I'd like to thank my supervisor Andreas Leitner who was never tired of discussing with me and clarifying things. Most design decisions, I made based on his council and he ensured that I never lost the thread on my way. He also helped to get the report that you hold in your hands into the right form, by investing hours on cross-reading and giving me advice.

Emmanuel Stapf of Eiffelsoftware inc. provided many useful tips about how to implement requested features. It always seemed that ten minutes of his time are equal to two weeks of mine, undoubtedly he saved me from poking in the countless lines of code that Eiffel studio contains, speeding up the development time enormously.

I'd like to thank Prof. Dr. Bertrand Meyer for making this work possible at all, first by designing the language Eiffel, which is still a class of its own and of course by letting me work for his group.

All members of the software engineering group who never hesitated to help me out, whenever I was stuck.

My co students Arno Fiva and Marco Zietzling: Arno for working together with me on the integration of selective capture and replay into Cdd and Marco for providing me his LaTeXtemplate for this report.

Alex Orso, one of the authors of the original paper about selective capture and replay [13] for quickly answering my questions regarding the technique.

Finally I want thank my family for supporting me throughout all my years of study and my girlfriend for cheering me up during this thesis; it is them who made all this possible.

Contents

Chapter 1

Introduction

Debugging applications is very tedious work. One of the hardest steps to find a fault in a program is to reproduce the steps that lead to the program failure. Capture and replay aims at making the execution of a program repeatable, removing this burden from the developer. In order to be able to replay an application, it is necessary to isolate the deterministic part of the application that needs to be replayed (observed part) and the non-deterministic parts, like user input, storage or network (unobserved part). Once the behavior of the unobserved part can be captured and replayed, it is also possible to replay the observed part. Although capture and replay is commonly known as capturing mouse and keyboard events, there are approaches that go further and capture events from all interactions with the environment. All these approaches have one thing in common: They have a fixed border between observed and unobserved part.

Selective capture and replay [13] makes it possible to individually define the observed part. Because the information flow is not equally intense between all parts of a program, this offers the possibility to optimize the amount of information that needs to be captured. To reduce the amount of captured data, selective capture and replay uses a special technique: whenever an object passes the border, only its type and a unique identifier is recorded, unless the passed object is from a basic type. It is possible to show, that this suffices to replay the observed part. This technique ensures that the amount of recorded data is linear to the number of variables passed through the border, in contrast to other techniques which capture the whole data that is passed

We have developed a selective capture and replay framework for Eiffel. It allows capture and replay functionality for whole applications with some changes in the runtime and modifications in the libraries that are significantly smaller than for traditional approaches.

The original implementation for selective capture and replay was made for Java. The technique proposed in the original paper can not be completely applied to Eiffel because some language features differ between Eiffel and Java. Some of these differences make an implementation easier, for example it is not possible to change attributes of other classes, and some make it harder, for ex-

ample multiple inheritance. Selective capture and replay relies on code instrumentation to capture accesses across the border. The original implementation instruments method and attribute accesses on the caller (client) side, whereas we instrument code at the callee side. The instrumentation our modified compilers adds to the program is the same for both the capture and the replay run. Hence no recompilation between the two phases is necessary. Depending on the program, compilation times can last for tens of minutes up to hours. For some capture and replay applications a recompilation can be prohibitive, therefore this feature is necessary. The original implementation instruments the program at the bytecode level and consequently doesn't need a complete recompile when the changing the code instrumentation.

This thesis provides an implementation that shows the feasibility of selective capture and replay in Eiffel, rather than the possible performance. Nonetheless, some performance measurements will be presented, based on a simple example application.

Chapter 2

Related Work

2.1 Overview

Because replaying application runs is important in order to be able to debug or test applications, there exist many techniques that implement capture and replay. One of the basic steps in order to be able to capture and replay an application is to distinct between the applications deterministic core (the *observed part*) and the non-deterministic environment like user input, network or external storage (the *unobserved part*). Capture and replay techniques capture the interactions between observed and unobserved part during the *capture phase* so that they can replay these interactions on the observed part during the *replay phase*.

Different implementations make different assumptions about what is deterministic and what changes its behaviour throughout different runs of the application. Therefore they define different portions of the program as observed and unobserved part. In the following we will categorize the different implementations based on which part is defined to be unobserved.

2.2 Capturing User Input

The best known capture and replay technique is the capturing of user input, which is most often used for GUI testing. Here, keyboard and mouse events are considered to come from the unobserved part. The exact location where these events are captured may vary (inside the application or through the operating system), but the advantages and limitations mostly stay the same. Usually, capture and replay of user input is used for regression testing. In that setup, the tester executes a sequence of actions on the application's GUI, while capturing is enabled. These recorded actions can be replayed on the application in order to check if there were regressions. Abbot [1] is an example of such a tool, although it offers more features than only capture and replay of user interactions.

Compared to our implementation, the capturing of user inputs has many

drawbacks due to its simplicity. Because only the user input is considered to belong to the unobserved part, it can only replay programs that interact with other parts of the environment (like file system and network) as long as these interactions are deterministic. One strategy is to restore the environment before every run of the application, but this is not always easy (e.g. if a random generator is used). If the assumption, that the observed part behaves deterministically does not hold, this results in an incorrect replay of the application. Because our implementation makes it possible to individually define the applications observed and unobserved part, it can also replay programs that depend on other parts of the non-deterministic environment than only the user input.

2.3 Capturing Interactions with Libraries

JRapture [14] uses an approach that has the potential of replaying more complex applications. JRapture is a tool for capturing and replaying Java applications in the field. In addition to capturing and replaying, it also offers a profiling interface that permits the program to be instrumented for profiling in the replay phase. JRapture uses the whole Java API as the unobserved part of the application. The technique is able to capture interaction between the Java API and the rest of the program by manual instrumentation of the Java API classes. JRapture supports multithreaded applications, but it can not guarantee a deterministic replay of concurrent applications.

Unlike our technique, JRapture relies on a manually modified version of the core libraries. Programs that need to be captured and replayed need to use a special version of the Java API. Therefore programs which interact with the environment through other mechanisms than the Java API (for example through the Java Native Interface JNI) can not be supported without additional manual instrumentations.

JRapture captures the complete information that is passed between the observed and unobserved part, in contrast to our approach that only captures necessary information.

2.4 Capturing Interactions with the Scheduler

The techniques presented so far did not consider thread scheduling as another source of non-determinism. DejaVu [6], a capture and replay tool for Java, considers thread scheduling as its only source of non-determinism, thus thread scheduling belongs to its unobserved part.
It is not easy to instrument the scheduler in order to detect thread switches, because threads are often scheduled by the operating system. DejaVu therefore introduces the concept of *logical thread schedule* which is a simplified version of the real thread schedule (the *physical thread schedule*). The *logical thread schedule* contains enough thread schedule information to reproduce the execution behaviour of the program under the assumption that the thread schedule is the only source of non-determinism. By detecting some critical events during

capture phase such as accesses to shared variables, and synchronization events, DejaVu is able to deduce the logical thread schedule.

DejaVu concentrates on enabling a determistic replay of multi-threaded programs that do not have any other source of non-determinism than the thread scheduler. Our technique does not take the non-determinism of the thread scheduler into account, it concentrates on all other interactions with the environment instead. DejaVu and selective capture and replay, as described by Joshi and Orso [13, 8], are othogonal and could be combined in order to allow replays of general multi-threaded programs.

2.5 Selective Capture Replay

In contrast to other techniques, *selective capture and replay* [13] implemented for Java, offers the possibility to make an own definition of observed and unobserved part of the system. Observed and unobserved part are both defined as a set of classes. The technique captures interactions between observed and unobserved part using automated code instrumentation.

There are two implementations of selective capture and replay: SCARPE, the original work of Joshi and Orso [13] and JINSI by Orso, Joshi, Burger and Zeller [12], which minimizes failure inducing component interactions, that were captured using selective capture and replay.

We transfer the concept from the original paper [13] to Eiffel; our implementation instruments code in a different way and adds some Eiffel specific modifications.

Chapter 3

Preliminaries

In this chapter we first present an example application that will be used throughout the report. Then we will summarize the paper about the original implementation of selective capture and replay by Joshi and Orso [13].

3.1 Example Application

Consider a customer that wants to withdraw money from his account. He interacts with the user interface at an ATM (Figure 3.1). This user interface can be implemented in different ways (textual or graphical). The transactions that the customer executes at the user interface are first written to the ATM's journal and then dispatched to the bank.

One use case is money withdrawal by a customer via an ATM. The customer launches the withdrawal operation on the *ATM_UI*. The request is passed through the *ATM* to the *BANK* which then withdraws the money from the Account. The sequence diagram (Figure 3.2) shows how the classes interact with each other.

Note that the names of classes and methods conform to the style presented in Object Oriented Software Construction, second edition [11]. Even Java examples that refer to this application will use that style to keep the same class and method names.

3.2 Selective Capture and Replay

After giving the overview about the different capture and replay techniques in the related work chapter , our choice will be described here in detail. Currently there exist two implementations of selective capture and replay, both for Java: SCARPE [13, 8], the original implementation, and JINSI [12], which minimizes failure inducing component interactions, that were captured using selective capture and replay.

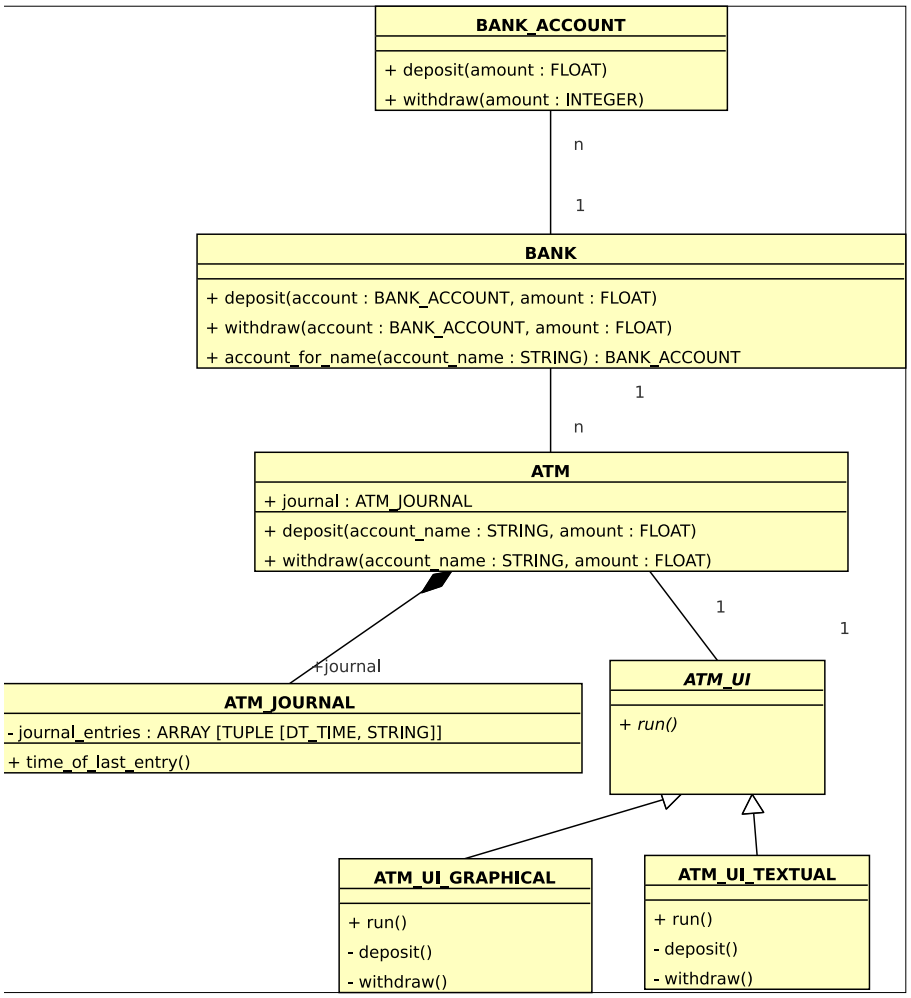

Figure 3.1: Class Diagram showing the example of a Bank and ATMs

This chapter summarizes papers from Joshi and Orso about the original implementation of selective capture and replay (SCARPE) [13, 8]. Where appropriate, some examples using the application described before are introduced.

3.2.1 Technique and Terminology

This technique lets the user choose the subset of the program that should be captured and replayed (*captured subsystem*, Figure 3.3). It then only captures the execution data of that subsystem, ignoring the other parts of the system. The relevant interactions between the captured subsystem and the outside world is captured in terms of events; those events are read during the replay phase in order to replay the corresponding interactions. Only the essential part of the information that traverses the border between the captured subsystem and the rest of the system is captured. This is a significant improvement over other ex-

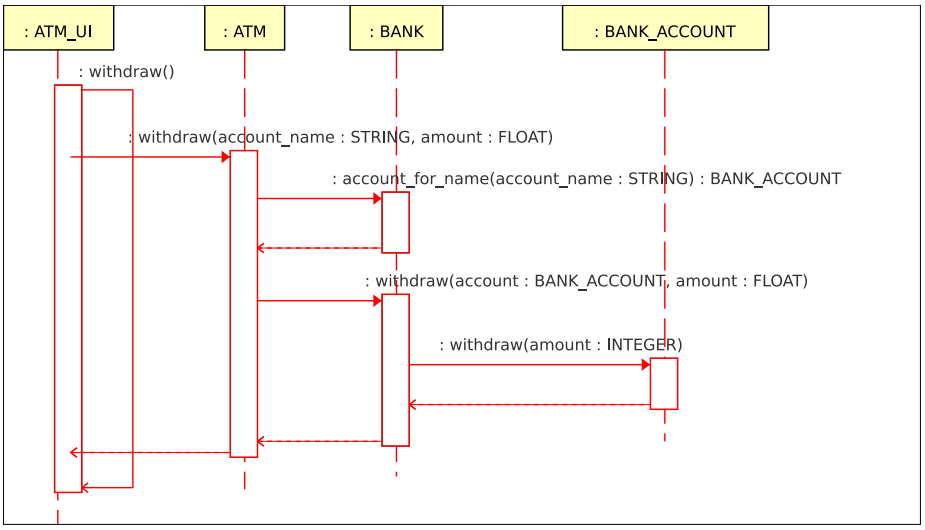

Figure 3.2: Sequence Diagram of a Withdrawal Operation Invoked by the Customer

isting techniques which capture all the data, especially when big datastructures are passed over the border and only a part of those datastructures is actually accessed.

Here are some terms that are used to describe selective capture and replay:

The observed set is the subsystem that was selected for capture and replay.

The observed classes are the classes in the observed set. Observed code is defined analogously.

Observed methods and fields are the fields and methods of the observed classes.

Unobserved set, *unobserved classes*, *unobserved code*, *unobserved methods* and *unobserved fields* are the corresponding terms for the part of the system that was not selected for capture and replay.

External code denotes either unobserved or library code.

Unmodifiable classes are the classes whose code can not be modified (e.g. some system classes such as *java.lang.Class*) due to constraints from the Java virtual machine.

Modifiable classes are all classes except the *unmodifiable classes*.

The technique is divided in two main phases: capture and replay. Figure 3.4 illustrates the setup of of these two phases in selective capture and replay. The *capture phase* takes place when the application is run for recording. Before the application starts, the boundaries of the observed set are identified and the

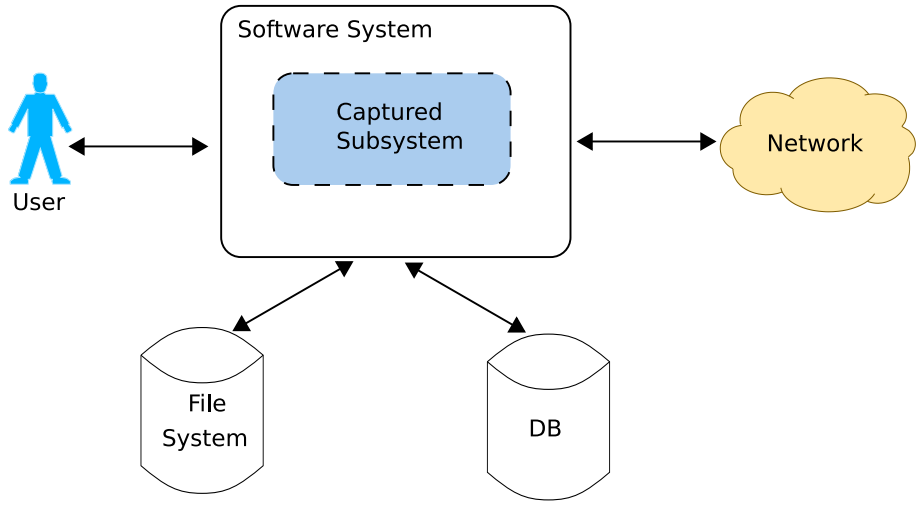

Figure 3.3: System Layout of the Example Application Showing the Captured Subsystem (derived from Joshi and Orso [13])

application is instrumented in order to be able to capture interactions between the observed set and the rest of the system. While the application runs, the instrumentation generates the events for these interactions and writes them to the *event log*.

In the *replay phase*, the technique generates the *replay scaffolding*. The replay scaffolding uses the event log to replay the events on the observed set. Replaying consists both of performing actions on the observed set (e.g. invocation of a method of the observed set) and consuming actions of the observed set (e.g. receiving a method invocation originally targeted to the unobserved set), thus the replay scaffolding acts both as a driver and stub.

3.2.2 Capture Phase

The capture phase is used to gather the necessary information to replay the application afterwards. In the following we will show how SCARPE [13] gathers that information.

Capturing Partial Information

To see the need for a solution of not capturing the whole data that flows over the border clearer, consider the case when a bank tries to find out the time of the last journal entry of an *ATM*. Assume that the classes *BANK* and *BANK_ACCOUNT* belong to the observed set and the rest of the application belongs to the unobserved set. The following instructions access the atm's journal and print the time of the last journal entry:

```
ATM_JOURNAL journal = atm.journal
```

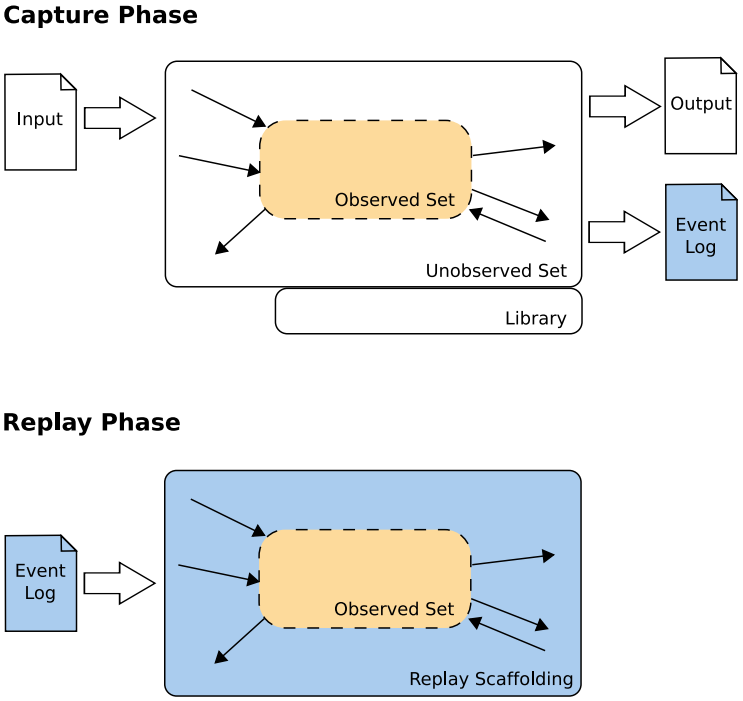

Figure 3.4: Capture and Replay Phase in Selective Capture and Replay (derived from Joshi and Orso [13])

```
print (journal.time_of_last_entry)
```

ATM_JOURNAL can contain a thousand of journal entries. In a traditional approach, due to the access to the journal, the whole data structure including all journal entries would be serialized each time the journal is accessed. The idea of selective capture and replay is to only write the referenced object that was passed (the journal) to the event log. When later *BANK* accesses the method *time_of_last_entry*, the result is again written to the event log, because the access crosses the border: from *BANK* (observed) to *ATM_JOURNAL* (unobserved). This reduces the size of the event log significantly, because only the the data that is used needs to be written down.

The technique for capturing only partial information instead of the whole data that flows into unobserved set relies on *object IDs*. An object ID uniquely identifies a class instance. SCARPE introduces the object ID into the program by adding an additional field to all modifiable classes. It further instruments the constructors of these classes so that the unique object ID is acquired from a global counter. For unmodifiable classes however, a reference map is needed to store the object ID of its instances. The object ID for instances of unmodifiable classes is acquired when it is looked up the first time. The reference map uses weak references so that the referenced objects can be collected by the garbage collector in order to avoid memory leaks.

Instead of recursively serializing the data that crosses the boundaries of the observed set, selective capture and replay only records a small part:

- Scalar values (values of a basic type) are fully captured.
- For objects, the technique only captures the object ID and the type. The type is used to restore the object again during replay phase.

Although the rules for this approach are simple, it is not obvious that they suffice to replay an application. Therefore we provide a short reasoning about why this works:

1) Instances of observed classes are fully populated during replay, because all actions from the unobserved set are replayed on them and the actions from the observed set are executed on them as in the original run.
2) During replay, accesses to methods or fields from observed code can either return instances of observed classes or instances of unobserved classes.
3) When an object of an observed class is returned, it is fully populated due to 1) and therefore its fields and methods can safely be accessed.
4) When an object of an unobserved class is returned, each further access to a field or method from observed code to that object results in the capturing of another boundary crossing event. Therefore the results of these accesses can be reconstructed during replay.
5) Therefore, it is assured that during replay, it is possible for the observed code to access the fields and methods like it did in the original run.

Interactions between Observed and External Code

Method calls are an elementary way to communicate between observed and external part of the application. The technique captures calls in both directions: from observed to unobserved code and vice versa. The following types of events are related to method calls:

OUTCALL events represent calls from observed code to unobserved methods.

INCALL events represent calls from unobserved code to observed methods.

OUTCALLRET events are triggered when returning from outcalls

INCALLRET events are triggered when returning from incalls

OUTCALL and INCALL events (*call events*) have the same attributes:

Receiver The receiver object

Method Called Signature of the called method.

Parameters The list of scalar values and objects passed to the method.

Capturing of call events involves capturing of scalar values and objects. They are written to the log as follows:

Scalar values are written verbatim to the log.

Objects are written by storing their type and object ID. For null values, the object ID is set to zero.

OUTCALLRET and INCALLRET events (*callret events*) have only one attribute: the returned value.

OUTCALL and OUTCALLRET events are captured by instrumenting the observed code. A probe is added to all calls to external methods.

To capture INCALL and INCALLRET events, proxy methods are introduced in a way that makes it unnecessary to instrument external code. It is ensured that calls inside the observed code don't use these proxy methods but the actual methods. Thus calls from observed code to observed methods can be executed without overhead.

Field accesses are another source of interaction between observed and external classes. Read accesses from unobserved to observed code don't influence the execution behaviour of the observed code and are therefore not captured. Thus three kinds of events are recorded:

OUTREAD events are generated when observed code reads the content of an unobserved field.

OUTWRITE events denote write accesses from observed code to unobserved fields.

INWRITE events are write accesses from unobserved code to observed fields.

OUTREAD, OUTWRITE and INWRITE have the same attributes:

Receiver: The object whose field is accessed.

Field Name: The name of the field that is accessed.

Value: The read or written value.

OUTREAD and OUTWRITE events are captured by adding probes in the observed code where fields of external classes are accessed. INWRITE events are captured in the same way, but here the unobserved code is instrumented whenever an observed field is written.

Exceptions are another interaction mechanism that needs to be considered for capture and replay. Because exceptions were not implemented in our Eiffel implementation of selective capture and replay, we will describe them only briefly. For exceptions two more types of events were introduced:

EXCIN events for exceptions that propagate from the external to the observed code.

EXCOUT events for exceptions that propagate from the observed to the external code.

EXCOUT events are captured by wrapping all observed methods with a *try--catch* block. By wrapping instructions that call an observed method with an exception handler, EXCIN events are captured.

3.2.3 Replay Phase

During the replay phase, the observed code is re-executed based on the event log that was gathered during the capture phase. Before replaying the observed code, selective capture and replay instruments the program based on the interactions between observed and external code.

Object Creation

In the replay phase, object IDs are treated in a different way. Now it is necessary to find the object that matches a given object ID, not reverse. The technique uses a reference map that maps object IDs to all available objects in order to resolve object IDs to objects. The technique extracts the object IDs from the event's attributes and then retrieves the associated object or creates it, if the object does not exist. How these objects are created depends on whether they are instances of observed or external classes.

Instances of External Classes: For external classes, *placeholder objects* are created. These objects have the correct type in order to support reflection in the observed code, for example the *instanceof* operator, but their state is irrelevant. After creation, the objects are registered in the reference map.

Instances of Observed Classes: Instances of observed classes are either created automatically by the observed code or there must be an INCALL event to the constructor in the event log. When replaying an INCALL to a constructor, an entry for the object is added to the reference map.

Null Values If the object ID is zero, *null* is returned.

Event Replaying

For the replay phase, a scaffolding is generated, which imitates the behaviour of the external code, and acts both as a driver and stub. Whenever control returns to the scaffolding, it checks whether the received event matches the next event in the log. If the events don't match (*out-of-sync events*), a message is displayed to the user, who can choose whether to ignore the discrepancy and continue, or to stop the replay. If the events from log and replay match, the event in the log

is consumed and replay continues.
Out-of-sync events can only occur, if the code was changed between capture and replay phase, for example because the technique is used for regression testing.

In contrast to INCALL, INWRITE, OUTREAD, OUTCALLRET and EXCIN, which are necessary to replay the observed code correctly, OUTCALL, INCALLRET, OUTWRITE, EXCOUT are not necessary to replay the observed code, but they can be used as oracles for regression testing.
For regression testing, the observed set is considered to be the system under test, which generates OUTCALL, INCALLRET, OUTWRITE and EXCOUT events as output, these events can then be compared to the recorded events of the capture phase - the recorded events are used as oracle.

INCALL Events To replay INCALL events, the method first retrieves the target object from the reference map. If the target object is not in the reference map, it must be either a call to a constructor or a static call. Then, it retrieves the parameters from the reference map, or creates the scalar values corresponding to the parameters. If an object parameter can't be retrieved from the reference map, a placeholder object is created. Afterwards, the technique can call the specified method with all arguments; the control flows to the observed code.

INCALLRET Events INCALLRET events are not essential for replaying the observed code. They occur as a result of INCALL events and are consumed by the replay scaffolding. To ensure a correct replay, the scaffolding checks if returned value conforms to the one of the next event in the event log.

OUTCALL Events All invocations to unobserved methods from the observed code are replaced by two instructions: (1) The call to a special feature in the scaffolding (*consumeCall*) and (2) The assignment of the result of the invocation. For example `TIME t = atm_journal.time_of_last_entry()`, with TIME and ATM_JOURNAL both being in the package *foo*, would be replaced by the following instructions:

```
Object tmp = scaffolding.consumeCall("foo/ATM_JOURNAL",
                24,/*object ID for atm_journal*/
                "time:()Lfoo/TIME",
                {} /*empty array of parameters*/
    );
TIME t = (TIME) tmp;
```

The scaffolding checks whether type, parameters, target and methodname of the next event from the log matches to the reported call. If the events match, replay continues with the next event, otherwise an error is reported to the user.

OUTCALLRET Events As these events occur as a result of OUTCALL events, they are treated within *consumeCall*. The method looks up the return value in the event log and retrieves it using the reference map.

OUTREAD and OUTWRITE Events To replay these events, accesses to unobserved fields are instrumented in the observed code, by calling associated methods from the scaffolding; *consumeRead* for OUTREAD events

and *consumeWrite* for OUTWRITE events. For example the instruction `ATM_JOURNAL journal = atm.journal`, with *ATM_JOURNAL* and *ATM* both belonging to package *foo*, is replaced by the following instruction:

```
ATM_JOURNAL journal = scaffolding.consumeRead ("foo/ATM"
    ,
                24,/*Object ID for atm*/
                "val")
```

The methods *consumeRead* and *consumeWrite* check if the event from the log matches the detected one. The method *outRead* furthermore returns the value associated with the event.

INWRITE Events In order to replay an INWRITE event, the scaffolding retrieves the target, the name of the field that should be modified and the value to be assigned. It resolves the target, which must exist unless it is a static field access. The value to be assigned is resolved as well or created if it does not exist yet. Finally the scaffolding sets the field of the target to the specified value.

EXCIN and EXCOUT Events To replay EXCIN events, the method *consumeCall* creates and throws a corresponding exception. This is possible, because EXCIN events can only occur during an outcall. EXCOUT events, which are generated by the observed code, are caught by the replay scaffolding and compared to the next event in the event log. If the exception matches the event from the event log, replay continues. Otherwise an error is reported to the user.

Assumptions

The following list of assumptions must hold for selective capture and replay:

- It is assumed that there is no direct access from unmodifiable classes to fields of observed classes.
- Because native code is not instrumented, Joshi and Orso also assume that there is no direct access from native code to observed fields.
- The technique can not guarantee the same thread schedule during capture and replay phase. Therefore it is assumed that the interleaving due to multi-threading does not change the behaviour of the observed code.
- The technique assumes that runtime exceptions occur deterministically. This is necessary, because exceptions generated in the observed code are consumed, but not replayed.

Special Cases

There are some special cases that need to be considered when implementing selective capture and replay:

Polymorphism and Dynamic Binding It is not always possible to determine whether an event is internal or external, when it depends on the dynamic type of the receiver. In these cases, it is necessary to add a runtime check to determine whether the receiver is an observed or external class, which is done using the *instanceof* operator. We assume that this case occurs at least in examples that use multiple interface inheritance, where it can not be assumed that the implementers of an interface are either all in the observed set or all in the unobserved set.

Inheritance If a class c is in the observed set, but not all of c's subclasses, creating an instance of any of c's subclasses would result in the creation of an instance of c. To avoid this issue, it is required that if a class c is added to the observed set, all subclasses of c are also in the observed set. We assume that problem described here is implementation specific, and could be avoided.

Reflection The technique handles most cases of reflection, however in some cases, additional instrumentation is required, for example if reflection is used in external code to modify fields of observed classes.

Access Modifiers In order to replay recorded executions, in some cases it is necessary to change access modifiers. We assume that this applies primarily to the *protected* modifier, which may prohibit access to observed classes from within the replay scaffolding, whereas some unobserved classes can access them.

Garbage Collection To allow the collection of unused objects, the technique must make sure that it does not keep references to unused objects. For example the reference map must use weak references to avoid memory leaks.

Finalize Calls to *finalize* are non-deterministic in Java, thus they can generate out-of-sync events during the replay phase. Therefore calls to *finalize* are treated in a special way: when receiving a call to *finalize*, the technique consumes the next matching entry in the event log, which is a call to *finalize*.

Chapter 4

Selective Capture and Replay for Eiffel

In the following we will describe how we implemented selective capture and replay for Eiffel with an emphasis on the differences to the original Java implementation. The chapter ends with a section that describes how to build our implementation from source and run an Eiffel example of selective capture and replay.

4.1 Differences to Existing Implementation

Selective capture and replay as described by Joshi and Orso [13] can not be directly applied to Eiffel. The core elements of the technique can be applied to Eiffel, but it is necessary to adjust some parts. In this section the changes to the original implementation and reasons for these changes will be discussed.

4.1.1 Language Aspects

Even though Java and Eiffel are both object oriented languages, there are differences between these two languages, syntax left aside. Eiffel offers a wider set of language constructs, many times trying to solve problems inherently object oriented, whereas Java reused solutions from its ancestors, mainly C++. One example for this difference are be the Eiffel basic types, which can be treated as objects with methods versus the Java basic types, which are no objects. This section focuses on the differences between Java and Eiffel that influenced our implementation of selective capture and replay.

Terminology

Eiffel has a nomenclature that differs from other programming languages. Here the Eiffel terminology is described according to Meyer's Book [11] and compared

to the one from Java.

Eiffel	**Java**
Attribute	Field
Query	Field or Method that has a return value
Routine	Method
Procedure	Method that has no return value
Function	Method that has a return value
Feature	Method or Attribute
ANY	Object (ancestor of all classes)

Multiple Inheritance

Eiffel is designed to support multiple inheritance, whereas in Java, only single inheritance is allowed. The original implementation assumes, that all subclasses of a class c are in the observed set iff c is also in the observed set. This makes it possible to statically decide whether an observed or an unobserved feature is accessed.

If this assumption is translated to multiple inheritance, it would not be possible to have a class C that inherits both from an observed class A and an unobserved class B (Figure 4.1).

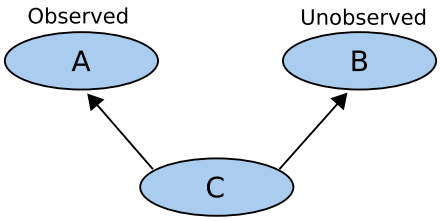

Figure 4.1: Conflict Between Observed and Unobserved Set Due to Multiple Inheritance

Multiple inheritance is extensively used in Eiffel, thus this assumption is too restrictive, because it leads to big clusters of classes that either all must be observed or all unobserved.

Although Java does not allow multiple inheritance, it supports multiple interface inheritance. Joshi and Orso [13] mention that they need to dynamically determine if the target of an event is observed or unobserved in certain cases. We assume that multiple interface inheritance with a class that both implements an observed and unobserved interface is at least one of these cases. SCARPE uses the *instanceof* operator to determine if an object is an instance of an observed class. For example if in our application *ATM_UI_TEXTUAL* is observed and the class *ATM_UI_GRAPHICAL* is unobserved, variables of type *ATM_UI* can reference an instance of an observed *or* an unobserved class. In that case, it is possible determine with the following Java code whether the actual object is an instance of an observed class:

```
if(ui instanceof ATM_UI_TEXTUAL){
   /*Instrumentation for instance of observed class*/
}else{
  /*Instrumentation for instance of unobserved class*/
}
```

We will present a solution that always determines dynamically, if an object is an instance of an observed class (*observed object*) or an instance of an unobserved class (*unobserved object*) and a proposal how to solve this problem without reflection.

Read Only Attributes

Eiffel strictly follows the *uniform access principle* [11]. The principle states that it should not be visible to the clients whether features are implemented through storage or computation. This ensures that the implementation of a feature can be changed from attribute to function or vice versa. One of the consequences of this principle is that clients of a class cannot write to attributes, as these attributes could be implemented as a function as well.

In Java, this principle is not ensured because clients of a class can write to their fields. For our implementation of selective capture and replay, this limits the events related to field accesses to OUTREAD. Both OUTWRITE and INWRITE can not happen, because write access to fields is restricted to their class, thus there can be no write accesses to fields from observed to unobserved classes or reverse.

4.1.2 Target Application

As a first application that could use selective capture and replay in Eiffel, Cdd, a tool for contract driven development [10] was chosen. Cdd allows programmers to extract test cases from failing program runs. The contracts that are present in the code are used as test oracles. Cdd is not always able to generate such a test case, due to different reasons:

Prestate extraction: The state before calling the failing feature (which is needed to generate a test case) is extracted using the state at the time of failure. Because not all instructions between feature call and failure can be undone, the extraction of the prestate is not always possible.

Non-determinism: If the failing feature reads values from sources that don't always return the same values (e.g. user input), it's not generally possible to run the test cases with the same result as in the failing run.

External state: When the feature relies on state that is not stored in Eiffel objects, for example in C structs, Cdd is not able to gather this state for the test.

All these limitations can be resolved using selective capture and replay:

Prestate extraction: By setting a breakpoint at the beginning of the failing feature, it is possible to extract the prestate during the replay of the application.

Non-determinism: When adding all non-deterministic classes to the unobserved set, it can be ensured that the replay of the run is deterministic.

External State: All classes that wrap external state can be added to the unobserved set. Thus interactions with these classes can be replayed without having to store the external state.

Prestate extraction using selective capture and replay requires an immediate rerun of the application under test. Because complete recompiles in Eiffel last for tens of minutes up to hours, we can not afford to instrument the application again before replaying it. Thus it is necessary to design the code instrumentation to be applicable for both capture *and* replay phase. We pushed this approach even further to allow the user to disable the whole instrumentation. This allows execution with minimal overhead if capture and replay is not needed, without recompiling the application.

4.2 Code Instrumentation

One of the key elements of selective capture and replay is code instrumentation, which is necessary to detect events that are relevant for the replay phase. In the following we will describe how application code is instrumented in order to detect events and how this instrumentation is inserted at compile time.

4.2.1 Outline

Due to differences between Eiffel and Java and a different use case, there are two main changes between the original implementation and our approach:

1. It is necessary to determine dynamically whether an object is observed or not.
2. It must be possible to switch between capture and replay phase without re-instrumenting the code.

The first requirement can be met by adding an additional query to every class so that it is possible to find out whether an object is an instance of an observed or an unobserved class. We ensure that every class has this query by adding it to *ANY*. In Section 4.3.4 we will discuss how this query can be implemented efficiently.

In order to be able to switch between capture and replay mode, the class *PROGRAM_FLOW_SINK* was introduced. Program flow events are put into the *PROGRAM_FLOW_SINK*. It is both the ancestor class of *RECORDER*, the class that contains the management code for recording events and *PLAYER*, the

class that provides the scaffolding for replaying events. The standard instance of a *PROGRAM_FLOW_SINK* is globally accessible. It is dynamically bound to an instance of *RECORDER* during capture phase and *PLAYER* during replay phase.

To create placeholder objects from unobserved classes, their original routine bodies are disabled during replay phase. When replaying, the routines only execute instrumentation code that invokes the scaffolding and takes care of the return value if necessary.

4.2.2 Routines

Whereas the original implementation captures routine calls by instrumenting the call site (in the client), we decided to instrument the calee (the supplier). This simplifies code instrumentation, because there is no need to make changes inside existing routine bodies; it is sufficient to add some code at the beginning and the end of the routine bodies.

The routine instrumentation can be divided into three parts (Figure 4.2): right at the beginning of the routine, code to detect call events is inserted. The original routine body is made conditional so that it is not executed for unobserved routines during the replay phase. Finally, code to detect callret events is inserted at the end of the routine.

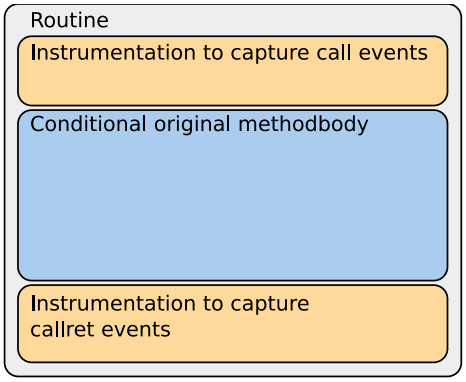

Figure 4.2: The Three Sections of Routine Instrumentations

To explain instrumentation in more detail, we describe each of these three parts using the query *account_exists* of the class *ATM* (Listing 4.1).

Listing 4.1: Original Code of Query *account_exists*

```
account_exists (account_name:STRING): BOOLEAN
    -- Is there an account with name 'account_name'?
  require
      account_name_not_void: account_name /= Void
  do
    Result := (the_bank.account_for_name (account_name) /= Void)
  end
```

Capturing Call Events

To capture call events, instrumented code, which is only executed when capture and replay is enabled, is inserted at the beginning of every routine. The code first detects whether the routine was called across the boundaries. If that is the case, it puts the associated event into the program flow (Listing 4.2).

Listing 4.2: Instrumentation Code to Detect Call Events

```
if program_flow_sink.is_capture_replay_enabled then
  program_flow_sink.enter
  if program_flow_sink.observed_stack_item /= is_observed then
    program_flow_sink.put_to_observed_stack (is_observed)
    program_flow_sink.put_feature_invocation ("account_exists", Current, [
        account_name])
  else
    program_flow_sink.put_to_observed_stack (is_observed)
  end
  program_flow_sink.leave
end
```

Note that all the inserted code is part of the selective capture and replay management code, which does not belong to the code of the original application, thus these instructions must not trigger further events for the event log. For example the creation of the manifest string "account_exists" should not create other call events on class *STRING*. To make sure, that this will not happen, the commands *enter* to disable capture and replay and *leave* to re-enable capture and replay were introduced.

To detect boundary crossing calls, the management code keeps an own stack that indicates whether the target objects of the calls on the call stack are observed or not. This stack is managed in parallel to the call stack; in the invocation instrumentation code, the result of the query *is_observed* is put onto the stack and in the exit instrumentation code, this value is removed again. Using this stack, it is possible to find out, whether the caller object is observed or not.

Conditional Routine Body Execution

To be able to switch between the original and the placeholder routine body, the original routine body of unobserved routines is only executed during capture phase (Listing 4.3). Placeholder routines are only needed for unobserved routines, thus the original body is always executed for observed routines . The query *is_replay_phase* indicates if the program is running in replay mode; it always returns false, when capture and replay is disabled, therefore the management code will always use the original routine body.

Listing 4.3: Conditional Methodbody

```
if (not program_flow_sink.is_replay_phase) or is_observed then
  Result := (the_bank.account_for_name (account_name) /= Void)
end
```

Capturing Callret Events

To capture and replay callret events, instrumentation code is inserted at the end of routines. Again, the code only puts an event into the *program_flow_sink* if the routine was called across the boundary. For functions, the code also restores the result to *program_flow_sink_.last_result*, which has only an effect for unobserved functions during replay phase, otherwise the query *last_result* returns the same value that was put into the *program_flow_sink* before. Thus functions can be used as placeholder functions without altering the instrumentation between capture and replay phase.

Listing 4.4: Instrumentation to Capture Callret Events

```
if program_flow_sink.is_capture_replay_enabled then
  program_flow_sink.enter
  program_flow_sink.remove_from_observed_stack
  if program_flow_sink.observed_stack_item /= is_observed then
    program_flow_sink.put_feature_exit (Result)
    Result ?= program_flow_sink.last_result
  end
  program_flow_sink.leave
end
```

Complete Routine Instrumentation

Listing 4.5 shows the routine after combining all three parts of the instrumentation.

Listing 4.5: Routine *account_exists* After Instrumentation

```
account_exists (account_name: STRING_8): BOOLEAN
    -- Is there an account with name 'account_name'?
  require
    account_name_not_void: account_name /= Void
  do
    if program_flow_sink.is_capture_replay_enabled then
      program_flow_sink.enter
      if program_flow_sink.observed_stack_item /= is_observed then
        program_flow_sink.put_to_observed_stack (is_observed)
        program_flow_sink.put_feature_invocation ("account_exists", Current, [
            account_name])
      else
        program_flow_sink.put_to_observed_stack (is_observed)
      end
      program_flow_sink.leave
    end
    if (not program_flow_sink.is_replay_phase) or is_observed then
      Result := (the_bank.account_for_name (account_name) /= Void)
    end
    if program_flow_sink.is_capture_replay_enabled then
      program_flow_sink.enter
      program_flow_sink.remove_from_observed_stack
      if program_flow_sink.observed_stack_item /= is_observed then
```

```
        program_flow_sink.put_feature_exit (Result)
        Result ?= program_flow_sink.last_result
      end
      program_flow_sink.leave
    end
  end
```

4.2.3 Attribute Accesses

Automated code instrumentation for attribute accesses could not be developed as part of this thesis, because the part of the Eiffel compiler code, that was intended to be used for instrumentation, turned out to be unsuitable for the task. Emmanuel Stapf of Eiffel Software inc. advised us to use the mechanism that generates wrapper code for attribute accesses to support expanded generic derivation reattachment (related to autoboxing). Unfortunately the compiler could not apply the mechanism to all attribute accesses and it was not possible to make the necessary changes in the compiler due to a lack of time.

Here is a sketch how the instrumentation code for an access to `atm.ui` could look like:

Listing 4.6: Instrumentation of an Attribute Access

```
if program_flow_sink.is_capture_replay_enabled and then is_observed and then (
    not atm.is_observed) then
  program_flow_sink.enter
  program_flow_sink.put_attribute_access("ui",atm, atm.ui)
  program_flow_sink.leave
end
ui := atm.ui
```

Again, the instrumentation code can be used for both capture and replay phase. The condition makes sure, that only OUTREAD events are captured and replayed. During capture phase, the *RECORDER* logs the corresponding OUTREAD event and during replay phase, the *PLAYER* ensures that the value of the attribute read in the next statement corresponds to the value that was read during capture phase, by setting the attribute to the value from the log.

The whole instrumentation needs to be put into a function that is accessed instead of the attribute, because the attribute can be accessed as part of a more complex expression:

```
a := other.function + other.attribute
```

It does not suffice to execute the instrumentation for the access to `other.attribute` one instruction before the evaluation of the expression, because `other.function` could have a side effect that changes `other.attribute`; although this violates the *Command-Query Separation Principle* [11], it can not be forbidden.

4.2.4 Attribute Manipulation from C Level

One assumption of the original selective capture and replay implementation is that native code never manipulates observed fields. Eiffel allows integration of C code into classes, using the *external* keyword. In order to write to fields of Eiffel objects, the C code needs the address of that object, which is achieved by using the *$* - operator.

To find out the classes whose attributes are written from native code, we searched for the *$* operator in the Eiffel base library. Only writes to {*STRING*} *.area* of type *SPECIAL [CHARACTER]* were identified to cause problems. Other than that, there were some write accesses to attributes of the class containing the C code, which is unproblematic because these accesses never cross the boundaries. Restricting the class *STRING* to be in the unobserved set was not an option, because it would imply serious performance penalties to applications that heavily use strings (e.g. a parser). To address this issue in a different way, the command *note_direct_manipulation* was added to the class *SPECIAL*, which needs to be called whenever unobserved C code writes to the fields of *SPECIAL*. Due to the lack of other use cases, the command was only designed to support *SPECIAL [CHARACTER]*. During the capture phase it writes down the new content of the object and when replaying the corresponding event, that content is restored; thus the objects are in the same state during replay as they were in the capture phase.

Manifest strings are another case where C code writes to strings, because the C code generated by Eiffel Studio allocates and initializes these strings directly. No automated instrumentation for manifest strings was developed, but it could be added by modifying the C macro that is used to generate manifest strings.

4.2.5 Compiler Integration

To insert the instrumentation code automatically, the compiler of Eiffel Studio was modified. The Eiffel Studio compiler has many backends: finalized C code, workbench C code, workbench melted code, Java bytecode and .Net CIL. Workbench code is produced in order to run it from Eiffel Studio; it allows faster compilation times with the drawback that it runs slower than finalized code. The generated code runs on a variety of platforms like Windows, Linux, several Unix operating systems, MacOS X and others [2]. To support as many backends as possible, one target was to keep the instrumentation code in Eiffel wherever possible, as the backend and platform independence is then guaranteed by the Eiffel compiler. To this end our modifications in the compiler change the abstract syntax tree (AST) before code generation, which is the approach that comes closest to changing the source code of the application directly.

Emmanuel Stapf advised us to make the modifications in the class *AST_FEATURE_CHECKER_GENERATOR*, which is used at the stage where the compiler checks the types of the program. Code instrumentation is inserted into the routines before their body is type-checked, in the feature *process_routine_as*. Our code generates additional AST nodes and replaces the original AST of the routine body.

A simple mechanism to exclude classes and routines from instrumentation was developed, because some classes like *PROGRAM_FLOW_SINK* need a special treatment to avoid management code invoking itself through instrumentation code. There are also some features that need to be instrumented in a special way, for example *SPECIAL.note_direct_manipulation* that needs to record the content of the instance during capture phase and restore that content during replay phase. To support classes and features that are not to be instrumented, two arrays are initialized with the names of exceptional classes and features. It is clear that at a later stage this needs to be controlled by a configuration file, to avoid recompiling Eiffel Studio, only because these exceptions changed.

4.2.6 Alternative Instrumentation

One drawback of our implementation is that the whole code is instrumented, whereas in the original implementation of selective capture and replay, only instructions that trigger an event are instrumented. Our implementation can result in a significant performance impairment, therefore we will outline an alternative or extension to our current approach to code instrumentation.

Andreas Leitner proposed to only instrument one of the two parts of the application: either the observed or the unobserved set. Assume in the following, that only the unobserved set is instrumented. With the existing instrumentation of routines, only OUTCALL and OUTCALLRET events would be captured. To capture INCALL and INCALLRET events, it is necessary to instrument all call sites in the unobserved routine bodies. When a routine call is detected, the instrumentation code first determines whether the call crosses the boundary, in which case the management code logs the corresponding event. Before every INCALL, the code sets a global flag that indicates whether code execution is currently in the observed or unobserved set. This will help to identify subsequent calls from observed code to unobserved routines.

4.3 Implementation

In this chapter, we will explain the implementation of the management code. For a quick overview over our implementation, we will first outline the architecture of the solution and then go through the different parts of the management code in the following sections.

4.3.1 Outline

Figure 4.3 shows the key classes of the implementation with the *PROGRAM_FLOW_SINK* as the entrance point to the management code. Program flow events are put into the standard instance of it, which all classes can access through the once function {*ANY*}.*program_flow_sink*. During capture phase, it is set to an instance of *RECORDER* and during replay phase, it is set to an instance of *PLAYER*.

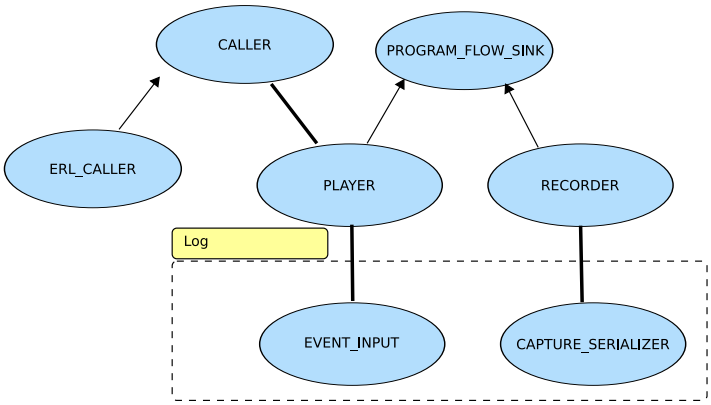

Figure 4.3: Overall Architecture of the Management Code

4.3.2 The Event Log

First it was intended to use XML as the format for the log files because it saves the task of writing a parser and the log file can be verified using XML Schema. For our parser, it was required to parse one event at a time, because the log files can become big when applications are captured over a long time. With the Gobo XML parser, it is not possible to request parsing of a single XML element at a time, there are only the options to use event based parsing or DOM parsing. Using the event based parser was mandatory, because the DOM based parser parses the whole log file and overly allocates memory in order to store all nodes of the DOM tree. The event based mode did not fit, because this required to invoke the replay scaffolding from the parser and not vice versa. One possibility would have been to put the parser into an own thread and block this thread whenever one element is buffered and not yet consumed by the replay scaffolding. Because multi-threading in Eiffel requires a compiler switch which causes alternative, incompatible libraries to be used, this was not an option; it would render the capture/replay mechanism unusable for single threaded applications. Therefore we decided to implement an own log file grammar (cf. Appendix A) with own parser and serializer to be able to incrementally parse the event log on demand.

The classes related to the log were designed to allow addition of other log formats in the future. For example *RECORDER* is using the deferred class *CAPTURE_SERIALIZER*, whose descendants can use their own log format (Figure 4.4). The deferred class offers a command for each event type to write the corresponding event to the log (e.g. *write_incall*) and *TEXT_SERIALIZER* implements these commands by writing out the events according to the grammar.

The parser is divided into the class *EVENT_PARSER* that reads from the log and *EVENT_PARSER_HANDLER* to handle events that were generated by the parser (Figure 4.5). {*EVENT_PARSER*}.*parse_event* parses the next event from the log and then calls the appropriate handler from *EVENT_PARSER_HANDLER*, for example *handle_incall_event*. *TEXT_EVENT_PARSER* is a

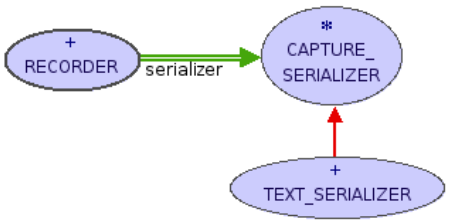

Figure 4.4: *CAPTURE_SERIALIZER* and *TEXT_SERIALIZER*

recursive descent parser [3] that parses the line based log file. To support other log formats, it is possible to implement other parsers that inherit from *EVENT_PARSER*. The player reads events from the class *EVENT_INPUT*, which uses the parser to read the events from the log.

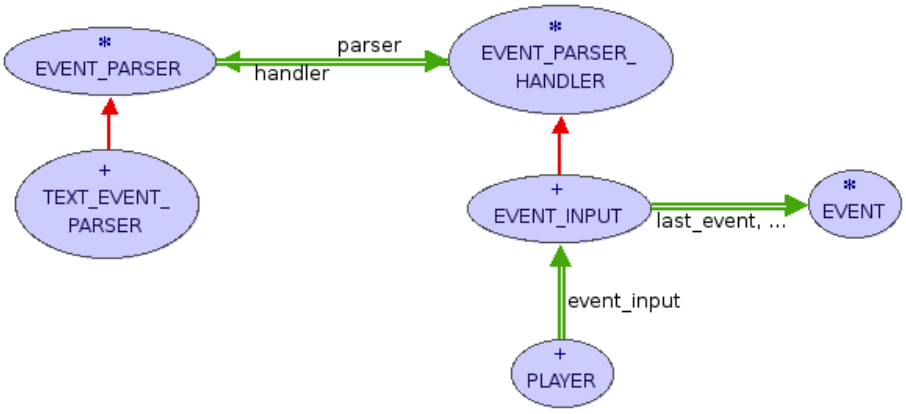

Figure 4.5: Architecture of the Event Log Parser

4.3.3 Object ID

Selective capture and replay relies on object IDs to be able to capture only partial data. To implement this for Eiffel, an object ID was added to the class *ANY* as an additional attribute, which is problematic, because the Eiffel runtime assumes a fixed size for all attributes of *ANY*, so it is only allowed to add routines to it. To circumvent this issue, the run-time was changed to allocate more space for each object. The object ID is stored at the end of the allocated memory to prevent disrupting existing code (Figure 4.6).

The class *SPECIAL*, which is able to store a variable number of elements (used for example for arrays), already makes use of this trick and stores the elements at the beginning and its two attributes *count* and *element_size* at the end of the allocated area. To avoid a clash with these attributes, the object ID in instances of *SPECIAL* is stored between the elements and these two attributes (Figure 4.7). Class *TUPLE* also uses a special object layout which renders the approach that was used for *ANY* unusable. Due to a lack of time, it was not

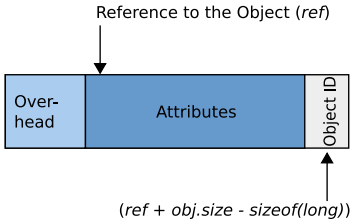

Figure 4.6: Object ID in the Object Layout of *ANY*

possible to add support for object IDs to the class *TUPLE*.

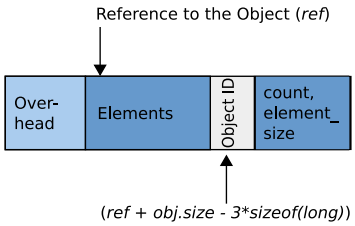

Figure 4.7: Object ID in the Object Layout of *SPECIAL*

Object IDs are lazily initialized, the first time the function *cr_object_id* is called, it retrieves a new object ID from the global counter *cr_global_object_id* and increases that counter. Because the object IDs are not allocated as regular attributes, it is necessary to access them via C code; there are two special C functions in *ANY* to read and write the object ID: *c_object_id* and *c_set_object_id*.

4.3.4 Query to Find out Whether an Object is Observed

Because of multiple inheritance, it is not generally possible to determine statically whether an object is observed or not. The original implementation of selective capture and replay uses the Java operator *instanceof* in special cases to determine the dynamic type of an object, in order to find out whether the object is observed. We decided to add a query to *ANY* that facilitates this task. This query can be implemented in different ways:

Solution 1: Look up in Hash Set

The first solution requires a single implementation of the query *is_observed* that uses a hash set to determine whether an object is observed. By looking up the generating type of the object in a hash set, which contains the names of all classes that are unobserved, the result can be calculated. Querying this implementation of *is_observed* generates subsequent routine calls, thus the overhead is big, considering the fact that *is_observed* is invoked three times for each routine call.

Solution 2: Repeated Redefinition of *is_observed*

Another idea is the implementation of a function that is redefined for each class. The function returns always the same value, that is, if the class is observed. The feature can not be a constant attribute, because it is not possible to redefine them, as constant attributes are resolved at compile time. Implementing the feature as a normal attribute would lead to a unnecessarily high memory consumption, because the value depends only on the class, not the instance.

It is not possible to make a standard implementation for unobserved classes in *ANY* and then only redefine it in all observed classes, because there will be conflicting redefinitions, where two ancestors of a class have different redefinitions of the same query [11] (Figure 4.8). Therefore this feature must be redefined in each class, which must be done automatically, because asking the developer to add a query to 4000 classes is not an option.

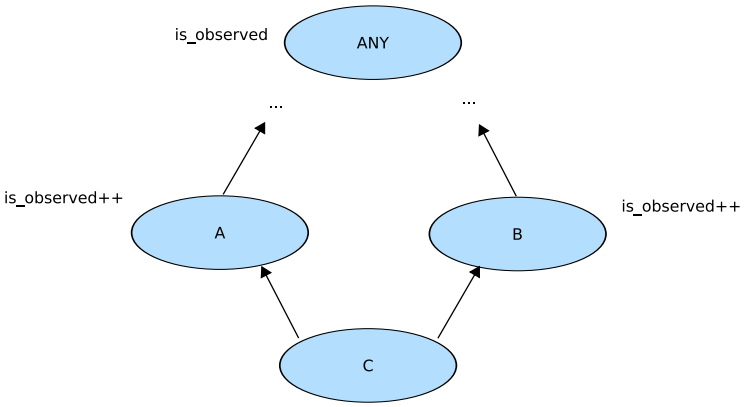

Figure 4.8: Example of Conflicting Redefinitions Of *is_observed*

Although this solution is more difficult to implement, it improves performance significantly, because accessing the query results in only one access to a feature instead of nested routine calls. It is possible that implementing this function as a once function can improve performance further.

Implemented Solution

Although the second solution has many advantages over the first solution, it was not chosen, because the implementation of the automatic instrumentation for it would have taken too much time. It is reasonable to replace our current implementation with this solution later, which will improve performance significantly.

4.3.5 The Recorder

The recorder receives the events that were passed to the program flow sink and records them to the log. To determine whether a call is an incall or outcall, it uses the *is_observed_stack* that indicates for all routines on the call stack whether they are observed or not. Then, the recorder writes the event to the log, using the *CAPTURE_SERIALIZER*.

4.3.6 The Player

The player is far more complex than the recorder, because it has to drive the application execution in contrast to the recorder which acts only as a passive sink for events (Figure 4.9).
Every time the player is invoked from the instrumentation code, it reads the next event from the *EVENT_INPUT* and compares that event to the one that has just occurred. *EVENT_INPUT* instantiates a different class for each event type. The events contain the same information that was written to the event log, for example the class *INCALL_EVENT* has three attributes: *target*, *feature_name* and *arguments*, a list of entities. An *ENTITY* is the representation of what is written to the log for objects (*NON_BASIC_ENTITY*) and scalar values (*BASIC_ENTITY*).
The player uses these entities to check values received from the instrumentation code or to retrieve the corresponding values using *ENTITY_RESOLVER* in order to pass them to the application code, for example when INCALL events are replayed.
The deferred class *CALLER*, an abstraction of reflection is used to call observed routines using the routine name, the target object and the arguments. Because Eiffel lacks native reflection support, the current implementation of *CALLER*, *ERL_CALLER* uses the reflection library generated by the Eiffel reflection library generator (Erl-G) [9]. The reflection library adds one reflection class for each class in the universe and causes all classes of the universe and the reflection library to be part of the system, which results in vast compilation times even for small projects. To bypass this issue, it is also possible to implement an own caller specific for a project, that is able to call all observed routines; but this approach is only reasonable when the observed set is small.

Resolving Entities

In the current implementation, creation procedures are instrumented in the same way as other routines, whereas the original implementation instruments creation procedures in a special way to maintain object IDs and register the objects in the reference map. To make sure that our technique can find objects to a given object ID, we use a different technique: everytime an object is passed to an unobserved routine, it is registered in the reference map. Objects only need to be resolved during replay phase, when the replay scaffolding needs to pass an observed object back to the observed set. Thus objects that never flow from the observed to the unobserved code are not interesting. During capture phase, the unobserved code must already have accessed an observed feature that

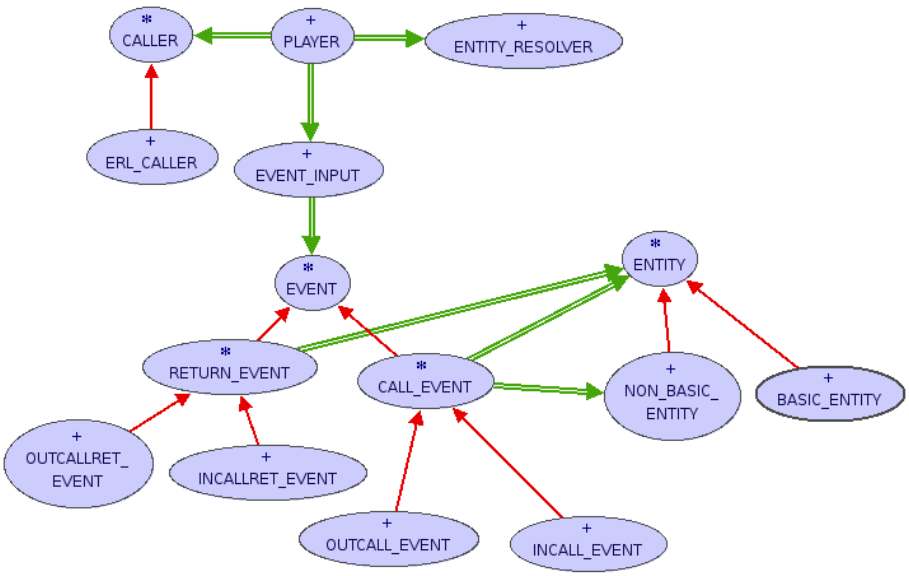

Figure 4.9: Architecture of the player

returns that observed object before it can pass the object back to the observed code; or it must have created this observed object. Therefore there is always an outgoing event, or the target of an INCALL that contains that object.

By the time of writing this report, we discovered a flaw related to this technique: because INREAD events are not captured, it is possible that unobserved code accesses an observed object without triggering an event. This could lead to the situation where unobserved code needs to pass an observed object back to observed code, without being able to resolve its object id, which would result in a failure during replay. To fix that problem it is either necessary to capture INREAD events, too or to fall back to the original implementation and register objects in their creation procedure. The original solution would result in a better performance for most applications, because objects are registered only once and not every time they're passed to the unobserved set.

Replaying of Events

Two routines of the player are invoked by the instrumentation code to notify it about a new event: *put_feature_invocation* and *put_feature_exit*. Using the *observed_stack*, the player then infers the type of event that has occurred. Depending on the event type, it needs to react in a different way:

OUTCALL An OUTCALL event is triggered, when an unobserved routine is called from observed code. The player is invoked and compares the next event in the event log with the actual event. If the type of the event and all parameters match, it registers all retrieved objects using the *ENTITY_RESOLVER* and consumes the current event. Then the player

starts to simulate the unobserved routine that was called, by executing all subsequent INCALL events that are in the event log. This is necessary whenever the unobserved code called observed routines, for example when the unobserved *ATM_UI* causes routine invocations on class *BANK* (Listing 4.7). The player only executes the calls, it does not consume the associated events as this is done by the instrumentation code in the corresponding observed routines.

Listing 4.7: Example Event Log that Requires Simulation of the Unobserved Routine

```
...
OUTCALL [NON_BASIC ATM_UI 5] run
INCALL [NON_BASIC BANK 1] account_for_name [NON_BASIC STRING_8
    7]
INCALLRET [NON_BASIC BANK_ACCOUNT 3]
INCALL [NON_BASIC BANK 1] deposit [NON_BASIC BANK_ACCOUNT 3]
    [BASIC REAL_32 "100"]
INCALLRET
...
OUTCALLRET
```

OUTCALLRET When the end of an unobserved routine that generated an OUTCALL event is reached, an OUTCALLRET event is triggered. During replay phase, this happens after the player finished the handling of the OUTCALL event. When player is then invoked again to handle the OUTCALLRET event, it reads the next event, resolves the result in the case of functions and returns to the unobserved routine.

INCALL When the instrumentation code triggers an INCALL event, the player already invoked the observed routine based on the INCALL event from the event log. Thus all the player needs to do is to consume the event and return to the observed routine.

INCALLRET The instrumentation code triggers INCALLRET events at the end of routines that caused an INCALL event. When the player receives an INCALLRET event, it compares the event to the next entry of the event log. If the events match, the player then registers the returned value and exits back to the observed routine.

4.4 Building the Example from Source

In this section the process of building an example with capture and replay support under Linux will be described. Installing the necessary tools and setting up the modified version of Eiffel Studio, that is able to instrument applications in order to capture and replay them, will take the biggest part of this explanation.

4.4.1 Building the Preliminaries

The first tools we need are the Eiffel Studio Tools [4]. These will be used in many setup scripts in the examples or tests from the repository. Install these tools according to the description on the webpage.

The delivery of the modified Eiffel Studio was built using revision 69201 of Eiffel Studio. Building a delivery with a later version of Eiffel Studio was not tested, so it might not work. After copying the delivery to /estudio/Eiffel60_gpl_69201, it can be activated:

```
activate_estudio 60_gpl_69201
```

Since the Eiffel compiler is available, Gobo [5] can be downloaded from svn and then bootstrapped (Revision 6001 was successfully tested).

```
svn co -r6001 https://gobo-eiffel.svn.sourceforge.net/svnroot/gobo-eiffel
    /gobo/trunk ~/capture_replay/gobo
export GOBO=~/capture\_replay/gobo
export PATH=$GOBO/bin:$PATH
$GOBO/work/bootstrap/bootstrap.sh gcc ise
```

It is assumed that all commands in the following steps are executed in the same session to keep the environment variables.
As all preliminaries are installed, Erl-G [9] can be downloaded and built. Revision 719 of Erl-G was tested together with selective capture and replay for Eiffel.

```
svn co -r719 https://svn.origo.ethz.ch/autotest/trunk/erl_g ~/
    capture_replay/erl_g
export ERL_G=~/capture_replay/erl_g
export PATH=$ERL_G/bin:$PATH
cd $ERL_G
export ISE_LIBRARY=$EIFFEL_SRC
geant install
geant compile
\title{Selective Capture and Replay for Eiffel
```

To build a delivery of the modified Eiffel Studio, execute these commands: (this will take a few hours).

```
cd ~/capture_replay/
mkdir es
svn co https://eiffelsoftware.origo.ethz.ch/svn/es/branches/
    capture_replay es
export EIFFEL_SRC=~/capture_replay/es/Src
cd es
geant -b $EIFFEL_SRC/scripts/build.eant build_es
```

Before an example can be built, the delivery that was just created needs to be set as default instance of Eiffel Studio

```
cd ~/estudio
ln -s ~/capture_replay/es/EiffelXX EiffelCR
activate_estudio CR
```

In order to make the created Eiffel Studio use a modified version of the runtime, it is necessary to recompile the runtime with modified CFLAGS. The new version of the runtime then needs to be installed in the delivery.

It is not possible to directly build Eiffel Studio with the modified version of the runtime, because the changes in the runtime are not compatible to Eiffel's store mechanism. This would render Eiffel Studio unusable, because it relies on this mechanism during the build process.

```
export CFLAGS='-DCAPTURE_REPLAY'
cd $EIFFEL_SRC
#build the runtime from scratch (clobber the old one)
geant -b scripts/build.eant compile_runtime
cd $ISE_EIFFEL/studio/spec/linux-x86/lib
rm *
cp $EIFFEL_SRC/C/run-time/lib* .
```

4.4.2 Building an Example

Now, all necessary tools are installed, the corresponding environment variables set and we can start to build an example.
First we need to add reflection support to the example project. Erl G will generate reflection classes for us. If the environment variables are correctly set, the geant script invokes Erl G automatically.
At the moment Erl-G does not support overrides because this feature is missing in the Gobo parser. Therefore it is necessary to override the necessary classes manually. There are two geant tasks that take care of this:

- *patch_elks* makes the manual override by copying the modified elks classes from $EIFFEL_SRC/library/base/capture_replay/elks_overrides to $ISE_LIBRARY/library/base/elks

- *unpatch_elks* restores the original state by copying the original elks classes from $EIFFEL_SRC/library/base/elks to $ISE_LIBRARY/library/base/elks .

```
cd ~/capture_replay/es/examples/capture_replay/command_line
geant install
```

The example can now be opened with the modified version of Eiffel Studio. Make sure that the CFLAGS are still set to '-DCAPTURE_REPLAY'. Otherwise it will not be possible to build the example.

An environment variable is used to control capture and replay of the instrumented application. Whether the application is captured or replayed depends

on the value of the environment variable *CR_MODE* at the application startup. The following values are supported:

not set When *CR_MODE* is not set, selective capture and replay will be disabled.

capture When the mode is set to *capture*, the program execution will be captured.

replay When *CR_MODE* is set to *replay*, the program will be replayed.

log_replay This mode replays the program and writes an event log in parallel. This event log has the same structure as the one that is written during capture phase. If the replay runs correct, the two logs are identical, therefore this feature can be used for testing.

Setting the environment variable controls the initialization of *program_flow_sink*. To set up all other necessary parameters for capture and replay phase, the creation procedure of the root class is used. This setup code needs to be inserted into applications that are to be captured and replayed. The setup code of the example can be used as a reference.

Chapter 5

Experimental Results

In this chapter we present the experimental results of our implementation and its limitations.

5.1 Performance Measurements

To measure performance of the implementation, we used a slightly modified version of our example application. We introduced *PERFORMANCE_TESTER_ATM_UI*, that inherits from *ATM_UI*. It simulates user input by depositing and withdrawing repeatedly (10'000 times in our tests) an amount from a bank account. For the measurements, all classes were observed by default, only the classes *ATM*, *PERFORMANCE_TESTER_ATM_UI*, *CONSOLE* and *STD_FILES* were unobserved.

We tested different scenarios for this application:

Capture Phase Running the application in capture mode.

Capture Phase - no events All classes are in the observed set, capture mode is enabled.

Replay Phase Running the application in replay mode, based on the log that was generated during the capture phase with regular observed and unobserved sets.

Capture and Replay Disabled Running the application with disabled capture and replay.

Uninstrumented Running the application without instrumentation, with no capture and replay management code involved.

5.1.1 Execution Times

The performance of the scenarios was measured using Eiffel Studio's built-in profiler, the results are in Table 5.1. The results show a significant increase

Setup	Total Execution Time
Uninstrumented	0.5s
Capture and Replay Disabled	4s
Capture Phase	9m
Capture Phase - no events	4m 25s
Replay Phase	27m 50s

Table 5.1: Total Execution times of the Tested Setups

in execution time, compared to the uninstrumented version; about 1000 times slower for the capture phase and more than 3000 times slower for the replay phase.
When comparing the two capture phase scenarios we can see that the generated events double the execution time. Therefore it is important to choose the boundaries between observed and unobserved set carefully.
The execution times show that the instrumentation code has an effect even when disabled, when comparing the scenarios *uninstrumented* and *capture and replay disabled*. The three additional *if* statements in each routine slow down the execution by the factor ten.

5.1.2 Possible Optimizations for the Capture Phase

In the following we will discuss how the performance can be optimized for the capture phase. Improving the performance is more important for the capture phase than the replay phase, because ideally developers should enable capturing whenever they are testing an application, which is only realistic, if the performance is acceptable. Table 5.2 shows an excerpt of the profiler output for the capture phase.

The feature is_observed

During capture phase, more than 48% of the time is spent in the feature *has_object* of class *UNOBSERVED_SET*, which is called from {*ANY*}*is_observed*. As written in the section about this feature (Section 4.3.4), this implementation needs to be replaced by a better-performing, which should save at least 48% execution time, because the execution of the feature *is_observed* itself,which uses the class *UNOBSERVED_SET* is not included in the 48%.

The class TEXT_SERIALIZER

Another big amount of time (43%) is spent in the class *TEXT_SERIALIZER*, which writes the events to the event log. This class can be further optimized by reducing the number of strings it allocates and optimizing the feature *is_basic_type*, which takes 10% of the overall time. This feature is implemented highly inefficient by using a look up table to determine whether the class name

Feature	**Calls**	**total time [s]**	**time [%]**
RECORDER.enter	1890359	0.88	0.16
RECORDER.leave	760147	0.38	0.07
RECORDER.observed_stack_item	760146	1.3	0.24
RECORDER.put_feature_exit	80019	52.32	9.74
RECORDER.put_feature_invocation	80019	198.95	37.05
RECORDER.put_to_observed_stack	380073	0.76	0.14
RECORDER.remove_from_observed_stack	380074	0.71	0.13
TEXT_SERIALIZER.basic_types	210041	0.08	0.01
TEXT_SERIALIZER.is_basic_type	210041	61.71	11.49
TEXT_SERIALIZER.print_debug	770172	6.46	1.2
TEXT_SERIALIZER.program_flow_sink	7711689	3.14	0.58
TEXT_SERIALIZER.write	610134	13.78	2.57
TEXT_SERIALIZER.write_arguments	80019	73.78	13.74
TEXT_SERIALIZER.write_basic	40009	12.67	2.36
TEXT_SERIALIZER.write_call	80019	177.1	32.98
TEXT_SERIALIZER.write_endline	160038	8.64	1.61
TEXT_SERIALIZER.write_incall	80018	180.64	33.64
TEXT_SERIALIZER.write_incallret	80018	51.98	9.68
TEXT_SERIALIZER.write_non_basic	170032	117.67	21.91
TEXT_SERIALIZER.write_object	210041	195.02	36.31
TEXT_SERIALIZER.write_return	80019	48.26	8.99
UNOBSERVED_SET.has_class_name	1130212	191.32	35.63
UNOBSERVED_SET.has_object	1130212	260.87	48.58
UNOBSERVED_SET.program_flow_sink	6781275	2.78	0.52

Table 5.2: Excerpt from the Profiler Output for the Capture Phase

represents a basic type.
The time spent in *TEXT_SERIALIZER* is proportional to the number of events that occur, in our example 160000. The number of generated events depends on the choice of observed and unobserved set. In our example two classes that communicate frequently, *BANK* and *ATM*, are divided by the boundary between observed and unobserved set, therefore many events are generated. A wise choice of observed and unobserved set together with a faster implementation of *TEXT_SERIALIZER* will result in a significantly better performance.

5.1.3 Possible Optimizations for the Replay Phase

Not every time an application is captured, it will be replayed, too. In our use cases, an application will only be replayed when running a test or when trying to find a bug. Therefore optimizing the replay of applications is not as urgent as for the capture phase. Nonetheless we will point out some parts that slow down the replay of applications, based on the profiler output for the replay phase (Table 5.3).

The class TEXT_EVENT_PARSER

More than 63% of the time is spent in the class *TEXT_EVENT_PARSER* for parsing the events in feature *parse_event*. The parser was not optimized and heavily uses string processing, which is further slowed down by the instrumentation code. Because the parser is a *recursive descent parser* [3], it frequently executes routine calls, as each symbol of the grammar is parsed in an own routine. By not instrumenting the *TEXT_EVENT_PARSER* it will be possible to make calls faster and thus speed up the parsing. With a more efficient implementation which minimizes the usage of strings, this will result in a significant performance improvement.

The class ERL_UNIVERSE_IMP

The class *ERL_UNIVERSE_IMP* returns the reflection class for a given object or object name. Our implementation uses it in the class *ERL_CALLER* to find the reflection classes in order to replay INCALL events on objects, using the feature *class_by_object* of *ERL_UNIVERSE_IMP*. Our test case triggers about 80'000 INCALL events, each resulting in a lookup for the reflection class during replay phase, using about 1% of the total execution time. Due to the implementation of the reflection library, calling a routine can result in the lookup of the reflection classes of the target's ancestors. This is necessary, because each reflection class is only able to call the features that were implemented in the corresponding class; for the routines implemented in its ancestors, the reflection classes of the ancestors are used to call them. This approach results in the heavy usage of the feature *class_by_name* with 980'000 calls and more than 7.5% of the execution time.

Native reflection support in Eiffel will certainly implement the features for calling routines more efficiently, which will result in a faster execution of the

Feature	**Calls**	**total time [s]**	**time [%]**
ERL_CALLER.call	80018	1650.2	98.92
EVENT_INPUT.read_next_event	160039	1064	63.78
PLAYER.consume_event	160039	1066.38	63.92
PLAYER.handle_incall_event	80018	1666.09	99.87
PLAYER.play	1	1668.26	100
PLAYER.put_feature_exit	80019	857.37	51.39
PLAYER.put_feature_invocation	80019	265.88	15.94
PLAYER.set_error_status_for_call	80019	26.56	1.59
PLAYER.simulate_unobserved_body	2	1668.26	100
TEXT_EVENT_PARSER.consume	790161	70.96	4.25
TEXT_EVENT_PARSER.end_of_line	2750567	23.76	1.42
TEXT_EVENT_PARSER.item	4600939	89.51	5.37
TEXT_EVENT_PARSER.matches	700151	61.88	3.71
TEXT_EVENT_PARSER.parse_event	160039	1062.22	63.67
TEXT_EVENT_PARSER.parse_line	160038	789.16	47.3
TEXT_EVENT_PARSER.parse_non_basic	170032	429.9	25.77
TEXT_EVENT_PARSER.program_flow_sink	42038769	16.76	1
UNOBSERVED_SET.has_object	1400263	295.3	17.7
FUNCTION.call_from_cr_management_code	40007	938.06	56.23
FUNCTION.fast_item	40007	937.36	56.19
PROCEDURE.call_from_cr_management_code	40011	493.26	29.57
PROCEDURE.fast_call	200049	496.91	29.79
SPECIAL.copy_data	3690774	47.83	2.87
ERL_CLASS_IMP_BANK.feature_	100014	18.35	1.1
ERL_CLASS_IMP_BANK.immediate_feature	100014	17.37	1.04
ERL_UNIVERSE_IMP.class_by_name	980228	128.07	7.68
ERL_UNIVERSE_IMP.class_by_object	80018	18.79	1.13
ROOT_CLASS.make	1	1668.27	100

Table 5.3: Excerpt from the Profiler Output for the Replay Phase

replay.

5.1.4 Comparison with the Results of SCARPE

In their paper about SCARPE, a tool for selective capture and replay for Java [13], Joshi and Orso present efficiency measurements for their implementation, using *JABA* (Java Architecture for Bytecode Analysis) as target application. Depending on the analysis that *JABA* needed to perform, the authors measured an overhead between 3% and 877% for the capture phase.

SCARPE instruments statically in most cases, it determines whether an instruction leads to a boundary crossing event based on the static type. When choosing the observed set wisely, this allows large parts of the application to run without any performance penalty, which is the reason why applications instrumented with SCARPE have such a low overhead, at most about 900% whereas our implementation has an overhead of 100'000%. As we discussed earlier, such a static instrumentation is not realistic for Eiffel, because it would strongly limit the choice for the observed set.

With a more efficient implementation, the overhead of our approach can be at least halved and by choosing the observed set in a way that minimizes the number of generated events, the overhead should drop further. But with the current instrumentation it will not be possible to reach the efficiency of SCARPE. Instrumentation of only one of the two parts of the application (either observed set or unobserved set), as presented in Section 4.2.6, promises to significantly raise the efficiency, because in that way, the uninstrumented set will execute without any overhead. When minimizing the instrumented set, the largest part of the application will be executed without overhead, which will result in a very efficient execution.

5.2 Contribution

The main parts of our technique were taken from the Java implementation of selective capture and replay from Joshi and Orso [13], however there are some aspects that differ from their implementation and are to the best of our knowledge new:

- Our implementation of selective capture and replay targets Eiffel as a language.
- We instrument applications in a way, so that the same executable can be used for both capture and replay phase. This makes it possible to replay an application immediately after it was captured, without recompiling it, which would take tens of minutes up to hours in Eiffel.
- Our technique instruments code at the callee side, whereas Joshi and Orso instrument code at the caller side.

- Our instrumentation code determines whether an object is observed or unobserved dynamically, and we proposed a solution to do this check with only a small performance overhead. This enables equal instrumentation in all cases of inheritance, fully supporting dynamic binding.

5.3 Limitations

Because of its limitations, the implementation that was made during this master thesis is to be seen as a proof of concept, only a small subset of programs can be captured and replayed. Here, we will provide a list of the known limitations:

Field Accesses The fact that there exists no automated code instrumentation for field accesses, limits the use of this implementation; however it has a weaker impact than it would have in Java, because in Eiffel, OUTREAD is the only type of field access that must be recorded in order to capture and replay an application. As a consequence of this limitation, only classes that don't read from unobserved fields can be put into the observed set.

Manifest Strings Manifest strings are directly initialized by the C code, that creates a string object and then directly writes the content of the manifest string into that object. Unlike the creation of the object, which is done using normal Eiffel routines, the initialization is directly done by the C code. This implies, that there is no instrumentation code invoked that could notice the change of the object's state.
If an observed manifest string created in unobserved code and then passed to observed code, this leads to a fault during replay, because no event is generated and the initialization of the string can not be replayed. By inserting manual instrumentation using `{SPECIAL}.note_direct_manipulation`, it is possible to solve this issue. However this solution is not satisfying as manifest strings are used in many places and requiring the developer to insert manual instrumentation in each of these places is not realistic. A better solution would be to change the C macros that are used by the generated code to create manifest strings, in order to invoke the management code whenever a manifest string is instantiated.

Selective Exports In Eiffel, a mechanism called *selective exports* [11] lets classes decide, to which classes their features are exported. This mechanism can be seen as a generalization of Javas *access level modifiers*. The problem arises whenever an observed class lets an unobserved class access a feature *f*, but prohibits the capture and replay management classes, especially class *CALLER*, from access to *f*. This leads to the situation where the event log contains an INCALL to the restricted feature, but *CALLER* is unable to call that feature during replay. A special case of selective exports are creation routines, that are often exported to *NONE* in order to restrict their usage exclusively to creation calls like `create foo.creation_procedure`, which does not care about export restrictions.

The case of restricted creation procedures can be solved by treating them specially and only calling them in context of object creation. The reflection library generated by Erl-G already makes this distinction and can be used in order to replay them correctly. However, this only solves a part of the problem, all other cases of restricted access are not addressed by this solution. The only reasonable possibility to address all problems that come with selective exports, is a native reflection support in Eiffel with the option to ignore access restrictions. Native reflection support would improve the compile times of capture and replay enabled applications significantly, by not doubling the amount of classes and not making all classes part of the system. For a wider applicable implementation of selective capture and replay, native reflection support is indispensable.

Language Features At the moment only a subset of the Eiffel language features is supported. In the following we will present a list of missing features:

Pre- and Postconditions In Eiffel it is possible to add to every routine a pre- and a postcondition. The precondition, initiated by the **require** keyword, defines what the caller of the routine must ensure in order to safely call the routine. The postcondition, initiated by the **ensure** keyword, defines what the routine ensures after execution under the assumption that the precondition was met. The developer can activate the checking of these conditions, which results in a check of the precondition before and postcondition after every routine execution.
Pre- and postcondition consist of a sequence of boolean Eiffel expressions, that contain function calls and attribute accesses. Thus when checking pre- and postconditions during application execution, additional events for selective capture and replay are triggered.
The technique must ensure that these function calls and attribute accesses do not take place during the replay phase if they are executed in the context of a unobserved routine, because it is required that unobserved code is not executed during replay phase.
Furthermore it is desirable that these events are all triggered in the context of the routine the corresponding pre- and postcondition belongs to. This implies, that the instrumentation code for routine invocation must be executed before the precondition is checked and the instrumentation code for routine exit is executed after the last postcondition is checked. Otherwise the events triggered by the assertion code is executed in the context of the caller.
At the moment selective capture and replay for Eiffel only works when checking of pre- and postconditions is disabled.

Class Invariants In contrast to pre- and postconditions that express the properties of a routine, class invariants, which are initiated by the **invariant** keyword, express properties of a class. Like pre- and postconditions of routines, invariants are a sequence of boolean Eiffel expressions. They need to hold before and after all exported routines; an exception to that rule are creation procedures that establish the invariant, therefore the class invariant generally does not hold before the execution of a creation procedure.

The checking of the invariants can be enabled by the developer, according to the Eiffel ECMA standard [7], they are then checked before and after every qualified routine call. As with pre- and postconditions of routines, this causes additional events during capture and replay phase, which is especially problematic for unobserved classes, which should not execute any code during replay phase. At the moment selective capture and replay for Eiffel only works with disabled invariant checking.

Exceptions The original implementation of selective capture and replay creates an event whenever exceptions are thrown across the boundary. The current implementation for Eiffel ignores exceptions, which can cause an incorrect replay of the application, as exceptions have an impact on program flow.

Expanded Types Variables of expanded type contain the object in contrast to regular variables that contain a reference to the object. Assigning a variable of expanded type to another variable (expanded or not) results in copying the object.
Expanded types were not incorporated in the implementation of selective capture and replay for Eiffel. In general, integrating support for expanded types is no problem, but there is one thing that must be considered: it is not possible to change an expanded object in another scope than the one it is defined in, because it is not possible to reference an expanded object. Therefore it is not possible to change an expanded object originated from application code in the management code. This makes the proposed solution for OUTREAD events, modifying the target object from management code before it is accessed, inapplicable for expanded objects. One solution to this problem is to change a copy of the expanded object and assign that copy back to the original object.

Agents Eiffel agents make it possible to wrap routines in objects. The current implementation of selective capture and replay for Eiffel has no support for agents.

Once Routines Once routines are routines that use the **once** keyword instead of the **do** keyword. As the name suggests, once routines are executed once, at least in single-threaded applications and if no special once key is defined (consult the Eiffel ECMA standard [7] for details).
The current implementation does not instrument once routines, hence they are not supported. When adding support for once routines, unobserved once routines must be treated specially, because it is possible that the first call of the once routine does not come from observed code. The technique must store the result nonetheless, because it is possible that a later call will come from observed code. Therefore the first call to once function must initialize its result correctly, because any subsequent call will return the same result.

Object IDs In the current implementation, object IDs are not supported for instances of class *TUPLE*. This was discovered a while after they were developed and is caused by the irregular object layout that instances of class *TUPLE* have.

It is possible that there are some more special cases left, although with *SPECIAL* and *TUPLE*, the usual suspects are treated.

When an object is copied using the feature *copy*, its object ID is copied, too. The correct behaviour in this case would be to request a new ID, because original and copy are two different objects and should therefore also have their own ID.

Another flaw regarding object IDs is that Eiffel's storable mechanism is not yet supported by the runtime with object ID support. Most probably this is also the reason, why Eiffel Studio using the modified runtime does not work properly.

Multi-Threading Selective capture and replay was not tested with a multi-threaded application, and some things were not designed with multi-threading in mind. One thing that certainly must be fixed in order to support multi-threading is the global counter for object IDs, which must be accessed using a mutex in order to avoid two objects, created by two threads, to have the same object ID.

When the whole management code is made thread safe, our implementation will still have the same limitation as SCARPE; it is required that multi-threading does not introduce any non determinism to the observed code.

Supported Compiler Backends At the moment, only frozen workbench code is supported, although most parts, like the automatically inserted instrumentation code also should work for other compiler backends, as they're implemented in pure Eiffel.

Chapter 6

Conclusions

In the following we will point out areas of future work for selective capture and replay for Eiffel and draw the conclusions of this thesis.

6.1 Future Work

A main target of future work on selective capture and replay for Eiffel must be to increase its performance. As pointed out before, it is necessary to change parts of the technique in order to reach an efficiency that can be compared with SCARPE.

Another area for future work are the implementation's current limitations. There are some limitations that restrict its usage to well prepared examples, for example the missing instrumentation of attribute accesses and the missing support for manifest strings. Making the current available features more robust is another area of work, for example the object ID support can be further improved so that it works under all circumstances. This requires additional example applications and test cases.

Native reflection support is crucial for the further success of selective capture and replay for Eiffel. The use of Erl-G is more to be seen as a workaround than a solution, because (a) there are some missing features, like access to selectively exported features and (b) it raises the necessary compile time of a project enabled for selective capture and replay to hours, making it necessary to have a dedicated machine to compile the projects. No developer will accept this increase of compile time on project he is working on, therefore selective capture and replay for Eiffel is doomed in productive environments, as long as there is no native reflection support in Eiffel. But native reflection support is something that this project can not influence, this feature must be provided by the language maintainers.

6.2 Conclusion

Our implementation of selective capture and replay for Eiffel shows that selective capture and replay is applicable to Eiffel. We found no fundamental problems, that could limit its implementation in Eiffel, neither for the implemented features nor the missing features.

The performance measurements show that optimizing the current implementation alone will not result in an efficiency that is comparable to SCARPE. It is necessary to change some basic techniques of the approach, for example incorporating the presented single-sided instrumentation (Section 4.2.6).

Appendix A

Log File Grammar

log ::= event*
event ::= call | return
call ::= calltype entity methodname entity* `%N`
return ::= returntype [entity] `%N`
calltype ::= `INCALL` | `OUTCALL`
returntype ::= `INCALLRET` | `OUTCALLRET`
methodname ::= identifier
attribute_name ::= identifier
entity ::= (object | value)
size ::= integer
object ::= `[NON_BASIC` typename object_id `]`
value ::= `[BASIC` typename `"` string `"]`
typename ::= identifier
object_id ::= integer
identifier ::= [A-Za-z]character*
string ::= character*

Bibliography

[1] Abbot java gui test framework. `http://sourceforge.net/projects/abbot`.

[2] Eiffel software inc. `http://www.eiffel.com`.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Patrick Ruckstuhl Bernd Schoeller, Andreas Leitner. Eiffel studio tools. `http://se.inf.ethz.ch/people/leitner/`.

[5] Eric Bezault and Others. Gobo eiffel tools and libraries. `http://www.gobosoft.com/`.

[6] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM Press.

[7] ECMA. *ECMA-367: Eiffel analysis, design and programming language*. June 2005.

[8] Shrinivas Joshi and Alessandro Orso. Capture and replay of user executions to improve software quality. Technical report, Georgia Institute of Technology, 2006.

[9] A. Leitner, P. Eugster, M. Oriol, and I. Ciupa. Reflecting on an existing programming language. In *Proceedings of TOOLS EUROPE 2007 - Objects, Models, Components, Patterns, (to appear)*, June 2007.

[10] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract driven development = test driven development - writing test-cases. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, September 2007.

[11] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[12] Alessandro Orso, Shrinivas Joshi, Martin Burger, and Andreas Zeller. Isolating relevant component interactions with jinsi. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 3–10, New York, NY, USA, 2006. ACM Press.

[13] Alessandro Orso and Bryan Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, may 2005.

[14] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 158–167, New York, NY, USA, 2000. ACM Press.