

Capture and Replay of User Executions to Improve Software Quality*

Shrinivas Joshi and Alessandro Orso
College of Computing
Georgia Institute of Technology
{orso|joshi}@cc.gatech.edu

ABSTRACT

Because today's software is increasingly dynamic and runs in heterogeneous environments, it is difficult to assess software systems outside the actual context in which they execute. Therefore, the results of in-house quality-assurance activities, such as testing, are often not representative of the way the software will behave in the field. To help addressing this problem, we present a technique for capturing and replaying partial executions of deployed software. Given an application, the technique allows for selecting a subsystem, capturing the dynamic interactions between such subsystem and the rest of the application, and replaying the recorded interactions on the subsystem in isolation. Our technique is designed to be efficient, in that we only capture relevant dynamic information and disregard all data that, although flowing through the boundary of the subsystem of interest, do not affect the execution. After describing our technique, we discuss several possible applications of the approach, including generation of test cases from users' executions and post-mortem dynamic analysis. In the paper, we also present SCARPE, a prototype tool that implements our technique for Java programs. Finally, we present an empirical evaluation of the technique performed using SCARPE. The results of the studies show the feasibility of the approach.

1. INTRODUCTION

The way in which we build software has changed considerably in the last decade. Software is increasingly dynamic, and its complexity is growing, together with the complexity of the environments in which it executes. Because modern software can behave very differently in different environments and configurations, it is difficult to assess its performance and reliability outside the actual time and context in which it executes. Therefore, the results of traditional quality-assurance activities—performed in-house, on developers' platforms, and using developer's provided inputs—are often not representative of how the software will behave in the field. In fact, the examples of incidents experienced by users because of untested behavior are countless.

To help addressing this problem, several researchers have proposed the collection of data from fielded systems and the

use of such data to augment in-house software engineering activities [2, 3, 7, 9, 11, 12, 16, 24]. Field data can provide unprecedented insight on the software's actual performance after deployment. In previous work, we collected data that supports this claim and provides evidence that, at least for the cases considered, executions in the field manifest a quite different behavior than in-house executions [12].

As a step towards addressing the limitations of purely in-house approaches, we present a technique and a technology for capturing and replaying user executions of deployed software. Overall, our technique works by (1) letting the user specify a subsystem of interest, (2) automatically identifying the boundary between such subsystem and the rest of the application, (3) efficiently capturing at runtime all of the interactions across this boundary, and (4) replaying the recorded interactions on the subsystem in isolation. Our technique is efficient because, for each execution, it disregards all data that, although flowing through the boundary of the subsystem of interest, do not affect its execution. Intuitively, the technique captures only the minimal subset of the application's state and environment required to replay the execution considered on the selected subsystem. Moreover, our technique can be straightforwardly applied to deployed software: it can either automatically generate a modified version of the software with the capture functionality built-in or be packaged together with the software and modify the code on the fly to enable the capture. The technique can be useful for many software-engineering tasks. In testing, for instance, the ability to capture and replay executions can allow for automatically getting test cases from users. Given a deployed program, we could capture executions in the field, collect and group them into test suites, and then use such test suites for validating the program in the way it is used. For another example, capture and replay would also allow for performing dynamic analyses that impose a high overhead on the execution time. In this case, we could capture executions of the un-instrumented program and then perform the expensive analyses off-line, while replaying.

1.1 State of the Art

Most existing capture-replay techniques and tools (*e.g.*, WinRunner: <http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/>) are defined to be used in-house, typically during testing. These techniques cannot be used in the field, where many additional constraints apply. First, all traditional techniques capture complete executions, which is generally infeasible in the field because it may require to record (and transfer) a huge vol-

*An early version of this work is described by one of the authors in a paper presented at WODA 2005 [14]. In this paper, we extend the capture-replay technique, describe the tool that implements the technique, discuss possible applications of the approach, and present a more extensive empirical study.

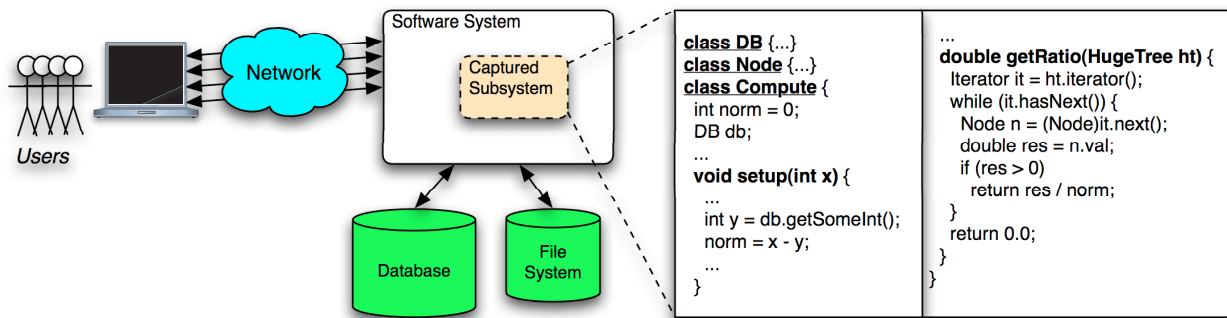


Figure 1: Example application.

ume of data for each execution. Second, existing techniques are typically based on capturing inputs to an application, which can be difficult and may require ad-hoc mechanisms, depending on the way the application interacts with its environment (*e.g.*, via a network, through a GUI). In fact, these techniques are typically focused on a specific domain (*e.g.*, GUI-based applications) and require a complex infrastructure and setup to be used. Third, there are issues related to side effects. If a captured execution has side effects on the system on which it runs, which is typically the case, replaying it may corrupt the system. Furthermore, the environment may have changed between capture and replay time, and there is no guarantee that the system will behave in the same way during replay. Fourth, existing techniques are not concerned with efficiency because they are not designed to be used on-line or on any kind of deployed software. Therefore, these techniques typically impose a considerable overhead during capture. Finally, capture-replay techniques that are designed to be used in-house do not take into account privacy issues. When capturing executions in the field, the data captured could contain confidential information that users may not want to be collected.

To the best of our knowledge, the only technique that addresses some of these issues is the one by Saff and colleagues [19], which aims to create mock objects for regression testing. However, these technique is also not designed to be used on deployed software and, thus, has different trade-offs and is not immediately applicable in this context (as discussed in Section 6).

To better illustrate the above issues, Figure 1 shows a software system, written in Java, that will be used as an example in the rest of the paper. The figure shows a networked, multi-user application that receives inputs from users and performs read and write accesses to both a database and the filesystem. The example is representative of situations in which capturing all of the information required to replay a whole execution would involve technical challenges (*e.g.*, collecting the data that flow from the users to the application and vice versa), storage problems (*e.g.*, the technique may have to record consistent portions of the database), and privacy issues (*e.g.*, the information provided by the users may be confidential). Using this kind of application as an example allows for stressing that many systems are complex and operate in a varied and complicated environment. However, the above issues would arise, to different extents, for most applications (*e.g.*, mail clients, word processors, web servers).

1.2 Advantages of our Approach

Defining a technique for capture and replay of fielded ex-

ecutions that accounts for practicality, privacy, and safety issues involves a set of challenges. Our technique is based on the concept of *selective capture and replay of executions*, is specifically designed to be used on deployed software, and addresses the issues with existing capture-replay techniques through a combination of novel technical solutions and careful engineering.

When practicality is concerned, the technique allows for limiting the volume of data that we need to record by suitably selecting the subset of the application for which we capture information.

Also, the technique addresses the problems represented by complex execution environments because it always captures (and replays) at the boundary between parts of the application. Therefore, no custom capture mechanism is required when different types of applications are considered.

When safety is concerned, our technique eliminates all side effects because it replays the subsystem in a sandbox—all interactions with the rest of the application and with the environment are only simulated during replay.

Our technique can mitigate efficiency issues because it captures executions of subsystems and, most importantly, because it captures only the minimal amount of information required to replay, as explained above.

When privacy is concerned, the use of our technique can help in two ways. First, it allows for excluding from the subsystem of interest those parts of the application that handle confidential information. Second, when this is not possible, our technique can be used to perform the replay on the users' machines instead of retrieving the captured execution and replaying in-house. For example, if the technique is used to perform expensive dynamic analyses on part of the application, it could capture executions for that part while users are running the application, replay them on an instrumented version when free cycles are available, and collect only sanitized results of the analysis.

1.3 Paper Organization and Contribution

In previous work, we presented an initial definition of our technique for selective capture and replay [13, 14]. The main contributions of this paper are:

- an extended and further detailed definition of our technique (Section 2),
- a discussion of a set of possible applications for the technique (Section 3),
- a description of SCARPE, our publicly available tool that implements the technique (Section 4), and
- an empirical study that shows the feasibility of our approach (Section 5).

2. CAPTURE AND REPLAY TECHNIQUE

In this section, we define our technique. For the sake of the presentation’s clarity, we limit the discussion in this section to the main technical characteristics of the work. We discuss several additional details separately, in Section 2.5.

2.1 Characteristics of the Technique

As discussed in the Introduction, defining a technique for capture and replay of fielded executions involves a set of challenges and issues of practicality, privacy, and safety. Our technique allows for overcoming these issues by providing a flexible and efficient way to capture and replay executions. More precisely, our technique has three main characteristics.

First, our technique captures and replays executions *selectively*. Users can specify the subset of the application that they are interested in capturing and replaying, and the technique only captures execution data for such subsystem. For example, considering Figure 1, we could specify that we are interested in capturing only the parts of the execution that involve the highlighted *application subset* (i.e., the parts of the executions that involve classes `DB`, `Node`, and `Compute`).

Second, our technique captures and replays executions in terms of events. During capture, the technique records every relevant interaction between the selected application subset and the rest of the system as an event with a set of attributes. During replay, the technique reads the recorded set of events and replays the corresponding interactions. For instance, in our example, each occurrence of the call to `it.hasNext()` in method `getRatio` would be recorded as a call event whose attributes contain information about object `it` and the called method.

Third, when recording events, our technique does not capture all the information that traverses the boundary between the selected application subset and the rest of the system. Instead, it captures only *partial data* and disregards the rest, which is of fundamental importance for the practicality of the technique. To illustrate, consider method `getRatio` of our example, which takes a parameter of type `HugeTree`, `ht`. Capturing complete information would require to store, for each call, a copy of object `ht`, which would result in an unacceptable space and time overhead (see Section 5). Optimizations of this process are possible, such as storing `ht` only when it changes between calls or adding `HugeTree` to the observed set, but they all have drawbacks. Detecting changes to `ht` imposes too much overhead and still requires several copies of the data structure to be stored. Conversely, adding `HugeTree` to the observed set may result in the set to become too large—in many cases, this approach would involve performing a transitive closure on the parameters types and including all the classes in the closure in the observed set.

2.2 Overview of the Approach

Because the characteristics of the programming language targeted by the technique considerably affect its definition, we define our technique for a specific language: Java. Although Java is our reference language, the technique should be generally applicable or adaptable to any object-oriented language that has the same features of Java (in particular, exception handling and multi-threading) or a subset thereof.

Terminology. In the paper, we refer to the selected application subsystem as the *observed set* and to the classes in the observed set as the *observed classes* (or code). *Observed*

methods and *observed fields* are methods and fields of observed classes. We define in an analogous way the terms *unobserved set*, *unobserved classes*, *unobserved code*, *unobserved methods*, and *unobserved fields*. The term *external code* indicates unobserved and library code together. The term *unmodifiable classes* denotes classes whose code cannot be modified (e.g., some system classes, such as `java.lang.Class`) due to constraints imposed by Java Virtual Machines. and the term *modifiable classes* refers to all other classes.

Our technique is divided in two main phases: capture and replay. Figure 2 informally depicts the two phases. In the rest of the section, we first provide an overview of the two phases and then discuss them in detail.

The capture phase takes place while the application is running (e.g., in the field or during testing). Before the application starts, based on the user-provided list of observed classes, the technique identifies the boundaries of the observed set and suitably modifies the application to be able to capture interactions between the observed set and the rest of the system. The application is modified by inserting probes (i.e., instructions added to the code through instrumentation) into the code and by adding proxy methods for each observed method that can be invoked by external code. When the modified application runs, the probes in the code suitably generate events for the interactions between the observed classes and the external code. The events, together with their attributes, are recorded in an *event log*.

In the replay phase, the technique automatically provides a *replay scaffolding*. The replay scaffolding inputs the event log produced during capture and replays each event in the log by acting as both a driver and a stub. Replaying an event corresponds to either performing an action on the observed set (e.g., writing an observed field) or consuming an action from the observed set (e.g., receiving a method invocation originally targeted to external code). Based on the event log, the replay scaffolding is able to generate and consume appropriate actions, so that during replay the right classes are created and the interactions among these classes are reproduced. The replay phase inputs the application, the list of classes in the observed set, and the event log. Based on the events in the log, it then (1) provides the right stimuli to the observed classes to replay the part of the execution that involves them, and (2) checks that the external events generated by the observed classes match the logged events. We now discuss the two phases in detail.

2.3 Capture Phase

As stated above, the capture phase works by (1) identifying all the interactions between observed and external code, (2) suitably instrumenting the application code, and (3) efficiently capturing interactions at runtime.

Before discussing the details of this phase, we need to introduce the concept of object ID. In the context of our technique, an *object ID* is a positive numeric ID that uniquely identifies a class instance. To generate such IDs, our technique uses a numeric *global ID* that is initialized to zero when capture starts. For modifiable classes, the object ID is generated by adding a numeric field to the classes and by adding a probe to the classes’ constructors. The probe increments the global ID and stores the resulting value in the numeric field of the object being created. Therefore, given an instance of a modifiable class, the technique can retrieve its object ID by simply accessing the ID field in the instance.

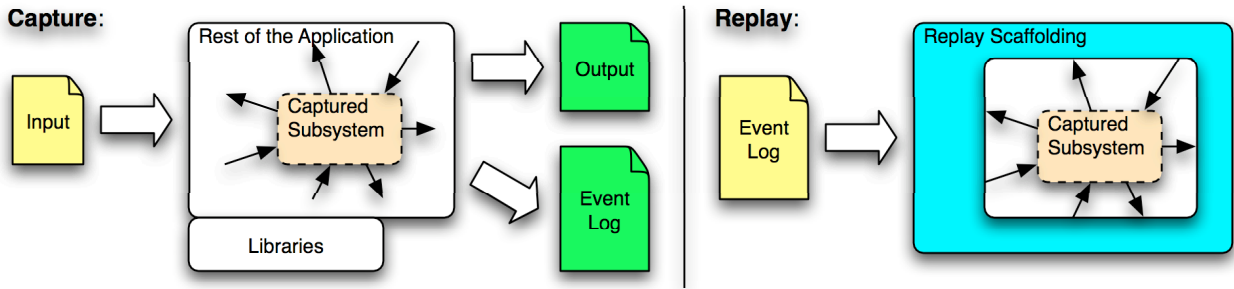


Figure 2: Overview of the capture-replay technique.

For unmodifiable classes, we associate IDs to instances using a *reference map*. The reference map contains information about how to map an object to its ID and is populated incrementally. Every time the technique needs an object ID for an instance of an unmodifiable class, it checks whether there is an entry in the reference map for that instance. If so, the technique gets from the map the corresponding ID. Otherwise, it increments the global ID and creates a new entry in the map for that instance with the current value of the global ID.

In the rest of the section, we first discuss how our technique can efficiently capture interactions by minimizing the amount of information to be recorded. Then, we discuss the different kinds of interactions identified by our technique, the corresponding events captured, and the approach used to capture them.

2.3.1 Capturing Partial Information

When capturing data flowing through the boundary of a subsystem (*e.g.*, values assigned to a field), the types of such data range from simple scalar values to complex and composite objects. Whereas capturing scalar values can be done inexpensively, collecting object values is computationally and space expensive. A straightforward approach that captures all values through the system (*e.g.*, by serializing objects passed as parameters) would incur in a tremendous overhead and would render the approach impractical. In preliminary work, we measured several orders of magnitude increases in the execution time for a technique based on object serialization. Our key intuition to address this problem is that (1) we only need to capture the subsets of those objects that affect the computation, and (2) we can conservatively approximate such subset by capturing it incrementally and on demand, without the need for sophisticated static analyses.

Consider again method `getRatio` in Figure 1 and assume that, for a given call, the first node whose value is greater than zero is the fifth node returned by the iterator. For that call, even if `ht` contains millions of nodes, we only need to store the five nodes accessed within the loop. We can push this approach even further: in general, we do not need to capture objects at all. Ultimately, what affects the computation are the scalar values stored in those objects or returned by methods of those objects. Therefore, as long as we can automatically identify and intercept accesses to those values, we can disregard the objects' state. For instance, in the example considered, the only data we need to store to replay the considered call are the boolean values returned

by the calls to the iterator's method `hasNext`, which determine the value of the `while` predicate, and the `double` values associated with the five nodes accessed.

Although it is in general not possible to identify in advance which subset of the information being passed to a method is relevant for a given call, we can conservatively approximate such subset by collecting it incrementally. To this end, we leverage our object-ID mechanism to record only minimal information about the objects involved in the computation. When logging data that cross the boundaries of the observed set (*e.g.*, parameters and exceptions), we record the actual value of the data only for scalar values. For objects, we only record their object ID and type. (We need to record the type to be able to recreate the object during replay, as explained in Section 2.4.) With this approach, object IDs, types, and scalar values are the only information required to replay executions, which can dramatically reduce the cost of the capture phase.

2.3.2 Interactions Observed–External Code

Method Calls. The most common way for two parts of an application to interact is through method calls. In our case, we must account for both calls from the unobserved to the observed code (*incalls*) and calls from the observed to the unobserved code (*outcalls*). Note that the technique does not need to record calls among observed methods because such calls occur naturally during replay. Our technique records four kinds of events related to method calls:

OUTCALL events, for calls from observed to unobserved code.

INCALL events, for calls from unobserved to observed code.

OUTCALLRET events, for returns from outcalls.

INCALLRET events, for returns from incalls.

OUTCALL and INCALL events have the following attributes:

Receiver: Fully qualified type and object ID of the receiver object. For static calls, the object ID is set to `-1`.

Method called: Signature of the method being called.

Parameters: A list of elements, one for each parameter. For scalar parameters, the list contains the actual value of the parameters, whereas for object parameters, the list contains the type of the parameter and the corresponding object ID (or a zero value, if the parameter is `null`).

OUTCALLRET and INCALLRET events contain only one attribute: the value returned. Analogous to call parameters, the attribute is the actual value in the case of scalar values, whereas it consists of the type of the value and the corresponding object ID if an object is returned.

To capture OUTCALL events, our technique modifies each observed method by adding a probe before each call to an external method. The signature of the method called is known statically, whereas the receiver's type and object ID and the information about the parameters is generally gathered at runtime.

To capture INCALL and INCALLRET events, our technique performs two steps.

First, it replaces each public observed method *m* with a proxy method and an actual method. The *actual method* has the same body as *m* (modulo some instrumentation), but has a different signature that takes an additional parameter of a special type. The *proxy method*, conversely, has exactly the same signature as *m*, but a different implementation. The proxy method (1) creates and logs an appropriate *INCALL* event, (2) calls the actual method by specifying the same parameters it received plus the parameter of the special type, (3) collects the value returned by the actual method (if any) and logs an *INCALLRET* event, and (4) returns to its caller the collected value (if any). In this case, all the information needed to log the events, except for the object ID and the return value, can be computed statically.

Second, the technique modifies all calls from observed methods to public observed methods by adding the additional parameter of the special type mentioned above. In this way, we are guaranteed that calls that do not cross the boundaries of the observed code invoke the actual (and not the proxy) method and do not log any spurious *INCALL* or *INCALLRET* event (these calls and returns occur naturally during replay).

Finally, to capture OUTCALLRET events, our technique again modifies the observed methods by instrumenting each call to an external method. For each such call, the technique adds a probe that stores the value returned by the call (if any) and logs it. As for the previous case, if the called method does not return any value, or the returned value is ignored by the code, a special value is used.

Access to Fields. Interactions between different parts of an application also occur through accesses to fields. To account for these interactions, our technique records accesses to observed fields from unobserved code and accesses from observed code to unobserved fields and fields of library classes. In the case of accesses from unobserved code to observed fields, we only record write accesses—read accesses do not affect the behavior of the observed classes and, thus, do not provide any useful information for replay. Further, unlike events generated in the observed code (*e.g.*, *OUTWRITE* and *OUTCALL* events), read accesses cannot be used as oracles because they are generated by code that is not going to be executed at all during replay. Our technique records three kinds of events for accesses to fields:

OUTREAD events, for read accesses from observed code to unobserved or library fields.

OUTWRITE events, for write accesses from observed code to unobserved or library fields.

INWRITE events, for modifications to an observed field performed by external code.

OUTREAD, **OUTWRITE**, and **INWRITE** events have the following attributes:

Receiver: Fully qualified type and object ID of the object whose field is being read or modified. As before, value `-1` is used in the case of access to a static field.

Field Name: Name of the field being accessed.

Value: Value being either read from or assigned to the field. Also in this case, the value corresponds to the actual values for scalar fields and to an object ID or zero (for `null`) otherwise.

To capture *OUTREAD* and *OUTWRITE* events, the technique first analyzes the observed code and identifies all the accesses to fields of external classes. Then, the technique adds a probe to each identified access: if the access is a read access, the probe logs an *OUTREAD* event with the value being read; if the access is a write access, the probe logs an *OUTWRITE* event with the value being written. The information about the field name is computed statically and added to the probes, whereas the information about the type and object ID is computed dynamically.

The method to capture *INWRITE* events is analogous to the one we just described for *OUTWRITE* events. The only difference is that the technique analyzes the modifiable external classes, instead of the observed ones, and instruments accesses to observed fields.

Exceptions. Exceptions are a widely used mechanism to report and handle erroneous situations. (A general description of the exception-handling mechanisms for the Java language can be found in Reference [10].) Exceptions too can cause interactions between different parts of an application. Moreover, interactions due to exceptions occur through implicit changes in the applications' control flow and are typically harder to identify than other types of interactions. For example, for the code in Figure 1, if the call to `ht.iterator()` in method `getRatio` terminated with an exception, the rest of the code in the method would not be executed. Not reproducing the exception during replay would result in a complete execution of the method, which does not correctly reproduce the recorded behavior. However, there is no point in `getRatio`'s code in which the fact that an exception has occurred is explicit.

To capture interactions that occur due to exceptions, our technique records two types of events:

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

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

EXCIN and *EXCOUT* events have only one attribute that consists of the type and object ID of the corresponding exception. To capture these events, our techniques uses Java's standard exception handling—it basically wraps relevant methods in suitable *try-catch* block.

To collect *EXCOUT* events, our technique wraps each observed method *m* with an exception handler that includes the entire method's body and handles exceptions of any type. (In Java, this instrumentation is realized by adding a *try-catch* block that includes the entire method and catches exceptions of type `java.lang.Throwable`.) The handler's code checks, by inspecting the call stack, whether *m*'s caller is an

external method. If so, it records the type and object ID of the exception, logs the corresponding EXCOUT event, and re-throws the exception. Conversely, if `m`'s caller is an observed method, the exception is still re-thrown, but is not logged as an EXCOUT event because it does not propagate to external code. (Note that the exception could still propagate to external code, if not handled by `m`'s caller. In such a case, however, the exception handler that wraps `m`'s caller would record the event.)

Similarly, to collect EXCIN events, our technique instruments all call sites in observed methods that call external methods. The technique wraps each such call site with an exception handler that also handles exception of any type. In this case, the handler's code gathers the type and object ID of the exception, logs the corresponding EXCIN event, and re-throws the exception. In this way, any exception generated as a consequence of an outcall would be suitably recorded.

Note that a single exception could result in multiple EXCIN and EXCOUT events, in the case in which it traverses the boundary between the observed and the external code multiple times.

2.4 Replay Phase

In the replay phase, our technique first performs two steps analogous in nature to the first two steps of the capture phase: it (1) identifies all the interactions between observed and external code, and (2) suitably instruments the application code. Then, the technique inputs an event log generated during capture and, for each event, either performs some action on the observed code or consumes some action coming from the observed code. In the rest of this section, we discuss how the replay phase handles the different logged events to correctly replay executions of the observed code.

2.4.1 Object Creation

In Section 2.3, we discussed how our technique associates object IDs to objects during capture. We now describe how object IDs are used in the replay phase, while generating and consuming events. Although we use a global ID and a reference map, analogous to the capture phase, the handling of IDs is different in this case. Unlike the capture phase, which associates IDs to objects flowing across the boundaries of the observed code, the replay phase extracts object IDs from the events' attributes and retrieves or creates the corresponding objects. Another difference between the two phases is that, during replay, all object IDs are stored in a reference map (not only the ones for instances of unmodifiable classes).

Instances of External Classes. Every time the technique processes an event whose attributes contain an object ID, it looks for a corresponding entry in the reference map. (The only exception is the case of object IDs with values zero or `-1`, which correspond to `null` values and static accesses, respectively.) If it finds an entry, it retrieves the object associated with that entry and uses it to reproduce the event. Otherwise, the technique increments the global counter, creates a placeholder object of the appropriate type (object IDs are always paired with a type in the events), and creates a new entry in the map for that instance with the current value of the global ID. A *placeholder object* is an object whose type and identity are meaningful, but whose state (*i.e.*, the actual value of its fields) is irrelevant. We need to preserve

objects' identity and type during replay for the execution to be type safe and to support some forms of reflection (*e.g.*, `instanceof`). Our technique uses *placeholder constructors* to build placeholder objects. For modifiable classes, the placeholder constructor is a new constructor added by our technique. The constructor takes a parameter of a special type, to make sure that its signature does not clash with any existing constructor, and contains only one statement—a call to its superclass's placeholder constructor.

For unmodifiable classes, our technique searches for a suitable constructor among the existing constructors for the class. In our current implementation, for simplicity, we hard-code the constructor to be used in these special cases (*e.g.*, `java.lang.Class`), but other approaches could be used.

Instances of Observed Classes. The case of observed classes is simpler. When replaying the incall to a constructor, the technique retrieves the object ID associated with the INCALL event, creates the object by calling the constructor (see Section 2.4.2), and adds an entry to the reference map for that instance and object ID. Note that, because of the way in which we replay events, instances will always be created in the same order. Therefore, we can use object IDs to correctly identify corresponding instances in the capture and replay phases, and to correctly reproduce events during replay.

2.4.2 Events Replaying

During replay, our technique acts as both a driver and a stub. It provides the scaffolding that mimics the behavior of the external code for executing the observed code in isolation. The replay scaffolding processes the events in the event log and passes the control to the observed code for INCALL, OUTCALLRET, and EXCIN events. When control returns to the scaffolding (*e.g.*, because an incall returns or an exception is thrown), the scaffolding checks whether the event received from the code matches the next event in the log. If so, it reads the following event and continues the replay. Otherwise, it reports the problem and waits for a decision from the user, who can either stop the execution or skip the unmatched event and continue. The case of events that do not match (*out-of-sync events*) can occur only when replaying events on a different version of the observed code than the one used during capture (*e.g.*, if the technique is used for regression testing).

Note that, whereas recording INCALL, INWRITE, OUTCALLRET, and EXCIN events (*incoming events*) is necessary to replay executions, the need for recording the events generated in the observed code (*outgoing events*) depends on the specific use of our technique. For example, if we use the technique to generate unit or subsystem test cases for regression testing, outgoing events are useful because they can be used as oracles. For another example, if we use the technique to compute def-use coverage off-line, we can disregard outgoing events.

We now describe the handling of the different events during replay.

INCALL Events. To replay INCALL events, our technique first extracts from the event its three attributes: (1) receiver, which consists of type and object ID, (2) method called, and (3) parameters.

Second, it retrieves from the reference map the instance

corresponding to the receiver’s object ID. In this case, the object is necessarily already in the map, unless the method called is a constructor or the invoked method is static. If the INCALL does correspond to a constructor, the technique calls the constructor to create the object and associates it with the object ID in the event. If the call is static, no object is retrieved.

Third, the technique scans the list of parameters. For each scalar parameter, it retrieves the value from the event. Conversely, for each non-scalar parameter, it retrieves the corresponding object using the object ID. This operation can have one of four possible outcomes. The object ID may be zero, in which case the value to be used is `null`. The ID may be `-1`, in which case no object is needed because the event corresponds to the invocation of a static method. The ID may be greater than zero, but not be in the reference map; in this case, the technique creates a placeholder object of the right type and adds it to the map. Finally, if the ID is in the map, the technique just gets the corresponding object. This object is either a placeholder object, if its class is not part of the observed set, or an actual object otherwise.

Finally, the technique calls the specified method on the object (or on the class, in the case of static calls) using the retrieved parameters. After the call, the control flows to the observed code. Note that passing a placeholder object (*i.e.*, an object with an undefined state) does not compromise the replay because all interactions of the observed code with external objects are suitably identified and intercepted by our technique.

INCALLRET Events. INCALLRET events occur as a consequence of an INCALL event and are consumed by the replay scaffolding. When the observed code returns after an INCALL, the scaffolding stores the return value, if any, and retrieves the next event from the event log. If the event is of type INCALLRET, the associated value is retrieved in the usual way (*i.e.*, as a scalar value or as an object ID) and compared to the value actually returned. If the values match, the replay continues with the next event. Otherwise, an error is reported and user intervention is required.

OUTCALL Events. OUTCALL events are also consumed by the replay scaffolding. Our technique instruments all observed classes so that each call to external classes is divided into two parts: the invocation of a specific method of the scaffolding (`consumeCall`), whose parameters contain information about the call, and an assignment that stores the value returned by `consumeCall`, if any, in the right variable in the observed code. For example, for the code in Figure 1, statement “`Iterator it = ht.iterator();`” would be replaced by the code (assuming that classes `HugeTree` and `Iterator` are defined in package `foo`):¹

```
Object tmp = scaffolding.consumeCall('foo/HugeTree',
    < object ID for ht >,
    'iterator:()Lfoo/Iterator'',
    <
empty array of parameters >);
Iterator it = (Iterator)tmp;
```

Method `consumeCall` retrieves the next event from the event log and checks whether the event is of type OUTCALL and the parameters match the attributes of the event. If so, the

¹Our technique actually operates at the bytecode level, and this example is just for illustration purposes.

replay continues with the next event. Otherwise, if either the event is of the wrong type or the parameters do not match (*e.g.*, the target of the call differs between capture and replay or a parameter of the outcall does not match the corresponding captured parameter), an error is reported to the user.

OUTCALLRET Events. To replay OUTCALLRET events, our technique extracts from the event the returned value, by retrieving it in the usual way based on its type (scalar or object), and simply returns that value. Because the handling of OUTCALLRET events is performed from within method `consumeCall`, and the technique modifies callsites in the observed code as described in the previous paragraph, the value returned, if any, is assigned to the right variable in the observed method that performs the outcall.

OUTREAD and OUTWRITE Events. To handle OUTREAD and OUTWRITE events, the replay phase instruments all observed classes so that each access to fields of external classes is replaced by a call to a specific method of the scaffolding: `consumeRead` for OUTREAD events, and `consumeWrite` for OUTWRITE events. For example, for the code in Figure 1, statement “`double res = n.val;`” would be replaced by the following code (assuming that class `Node` is defined in package `bar`):

```
double res = scaffolding.consumeRead('bar/Node',
    < objectIDfor n >,
    'val');
```

Method `consumeRead` retrieves the next event from the event log and checks whether the event is of the right type and the parameters match the attributes of the event. If so, it retrieves the value associated with the event and returns it. Otherwise, it reports an error to the user. Method `consumeWrite` behaves in an analogous way, but does not return any value because, in the case of OUTWRITE events, no variable in the observed code is modified.

INWRITE Events. To replay an INWRITE event, our technique first retrieves from the event attributes (1) the receiver object (if the accessed field is non-static), which represents the object whose field is being modified, (2) the name of the field being modified, and (3) the value to be assigned to the field. As usual, the value can be an actual scalar value, an actual object, a placeholder object, or null. Analogous to INCALL events, if the field is non static, the receiver object is necessarily already existent when the INWRITE event occurs. After collecting the information, the technique simply sets the value of the field in the identified object (or in the class, in the case of static fields) to the appropriate value.

EXCIN Events. Our technique replays EXCIN events by extracting from the event the object ID for the exception, retrieving the corresponding object, and throwing it. Like OUTCALLRET events, also the handling of EXCIN events takes place within method `consumeCall` because exceptions that propagate from external to observed code occur, directly or indirectly, during the execution of an outcall.

EXCOUT Events. The replay scaffolding consumes EXCOUT events by providing an exception handler that catches

any exceptions that may propagate from the observed code. The handler retrieves the next event from the event log and checks whether the event is of type EXCOUT and the exception thrown matches the exception that was recorded. If so, the replay continues with the next event. (In the case of an uncaught exception that never reenters the observed code and results in the termination of the program, there would be no further events in the log, and the replay would terminate.) Otherwise, an error is reported to the user.

2.5 Additional Technical Details

For space reasons, and to simplify the presentation, we purposely glossed over some of the technical details of the approach. In this section, we concisely discuss the most relevant ones.

2.5.1 Assumptions

Our technique works under some assumptions. *First*, we assume that there is no direct access from an unmodifiable class to a field of an observed class. Unmodifiable classes are typically in system libraries, so we expect this assumption to hold in most cases—libraries do not typically know the structure of the application classes. *Second*, because our current implementation does not instrument native code, we also assume that there is no direct access from native code to an observed field. Except for this case, our technique can handle native methods in external code like any other method. *Third*, we assume that the interleaving due to multi-threading does not affect the behavior of the observed code because our technique does not order “internal events” (*e.g.*, calls between observed methods), which occur naturally during replay. *Finally*, we assume that runtime exceptions occur deterministically. The technique consumes but does not replay exceptions generated in the observed code, which are expected to simply occur during replay.

2.5.2 Special handling of specific language features

Polymorphism and Dynamic Binding. In some situations, whether an event (*e.g.*, a call or a field access) is internal or external depends on the dynamic type of the event receiver. In such cases, our technique cannot statically decide the kind of instrumentation to use and must insert in the code a runtime check that determines whether the receiver of the event is an external or an internal class. The runtime check uses primitive `instanceof` to check the type of the receiver object and, based on the results, logs or disregards the event. This kind of check can be necessary for various types of events, such as OUTCALL, INCALL, OUTREAD, and OUTWRITE.

Inheritance. Inheritance can introduce subtle interactions between a class and its subclasses, due to polymorphism and dynamic binding. If the observed set contains a class *c*, but not all of *c*’s subclasses, creating dummy instances of any of *c*’s subclasses would result in the creation of an instance of *c*. Although it is possible to avoid this problem through additional instrumentation, we decided to address the issue by requiring, if a class *c* is added to the observed set, to add also all subclasses of *c*. This solution lets us simplify the instrumentation and has not been problematic or limiting in our experience so far.

Reflection. Our technique can handle most uses of reflection. However, in some cases (*e.g.*, when reflection is used in external code to modify fields of observed classes), addi-

tional instrumentation is required. For instance, to capture reflection-based field access events, the additional instrumentation intercepts calls to “getter” (*e.g.*, `getBytes(java.lang.Object)`) and “setter” (*e.g.*, `setBoolean(java.lang.Object)`) methods of class `java.lang.reflect.Field`.

Access Modifiers. In some cases, the technique must change access modifiers of class members to be able to replay recorded executions (which can be done without affecting the semantics of the observed code).

Garbage Collection. To account for garbage collection, our technique must ensure that it does not keep references to objects that would be otherwise garbage collected. For example, the reference map must use weak references (*i.e.*, references that do not prevent the referenced objects from being garbage collected) to avoid memory leaks.

Finalize Because calls to `finalize` are non-deterministic in Java, they can generate out-of-sync events during replay. Therefore, we treat calls to `finalize` (and method calls originated within a `finalize` method) in a special way: If a replay generates an out-of-sync event, before reporting an error the tool checks if the event is a call to `finalize`. If so, it consumes the first matching call to `finalize` in the execution log and performs the call to the corresponding `finalize` method.

3. POSSIBLE APPLICATIONS OF OUR CAPTURE-REPLAY TECHNIQUE

We discuss three applications of our technique: post-mortem dynamic analysis of users’ executions, debugging of deployed applications, and regression testing based on users executions.

3.1 Post-mortem Dynamic Analysis of Users’ Executions

The first application that we propose for our capture-replay technique involves the use of the technique for selectively capturing users’ executions and performing various dynamic analyses while replaying these executions. Being able to perform dynamic analysis on the users platforms would provide software producers with unprecedented insights on the way their software is used in the field. A perfect example of this situation is the identification of memory-related problems performed by tools like Valgrind [21]. These tools have been used very successfully in-house to identify such problems, but may miss problems that occur only in some specific configuration or for some specific runs. Unfortunately, the overhead imposed by these runtime memory-checking tools is too high for them to be usable on deployed software.

We define different scenarios for this application. A first scenario involves the collection of the users’ executions to replay them in-house. This scenario is ideal for the software producer, who would be able to perform many kinds of dynamic analysis, in house, on real (albeit partial) executions. However, collecting executions for some parts of a system may introduce serious privacy issues, and executions of those parts may be necessary for the analysis considered (*e.g.*, it is likely that the detection of memory-related problems will require to capture executions of large subsystems).

To account for this problem, we define a second scenario, in which executions are not collected in-house, but are instead stored locally, on the users’ machines. In this scenario, the replay and the dynamic analysis would occur on the

users' platforms, by leveraging free cycles (*e.g.*, at night). The only data collected in-house would be the results of the analysis, possibly further sanitized. This second scenario would allow for addressing privacy concerns and also for limiting the amount of data to be transferred over the net.

For both of the above scenarios, capturing all executions performed in the field could make the approach impractical if the amount of information to be stored, either in-house or remotely, is too high. Therefore, we define a third scenario, in which some criterion is used to decide which executions to collect. There are several criteria that could be used to this end, some of which may depend on the specific kind of analysis the software producer is interested in performing. For example, one criterion could be to collect only executions that terminate with an exception (or a crash) and discard normally-terminating runs. For another example, statistical approaches could be used to sample the executions according to some distribution. For yet another example, a technique that classifies executions (*e.g.*, [4, 6]) could be used to decide which executions to collect.

3.2 Debugging of Deployed Applications

The second application of the capture-replay approach that we propose is debugging of deployed application. Consider again the example in Figure 1. The example contains the following fault. If (1) the integer passed to `setup` has the same value as the integer returned by the call to `db.getSomeInt` within `setup`, (2) the value of field `norm` is not redefined, (3) method `getRatio` is called, and (4) predicate "`res > 0`" evaluates to true at least once, then the application generates a division by zero and fails. An execution that terminates with such failure could be arbitrarily long and involve a number of interactions between users, application, and database/filesystem. However, capturing only the execution of class `Compute` would provide enough information to locate and remove the fault.

In situations like the one of the example, our capture-replay technique could be applied to capture executions that can reproduce a failure in the field and send them to the software developer. As a possible scenario for this application, consider a program that is deployed in a number of instances, and for each instance a different subsystem is captured (*e.g.*, by partitioning the program in subsystems and assigning the capturing of each subsystem to one or more user sites). When a failure occurs at a given site, in a subsystem that is being captured at that site, the corresponding execution is saved for later debugging.

At this point, one possibility is to send the captured execution to the software producer's site, where it can be used for traditional debugging. Being able to debug failing executions collected in the field would be extremely useful in itself. However, another possibility is to perform (semi)automated debugging remotely, on the site(s) where the application fail. To this end, our capture-replay technique could be combined with an automated debugging technique (*e.g.*, delta debugging approach [25]) applied to captured executions. For example, delta debugging could be used to find a minimal set of interactions that lead to the failure and then send back to the software producer only this minimized set.

3.3 Regression Testing based on Users Executions

Regression testing (*e.g.*, [15, 17]) is performed on a modified

version of a program to provide confidence that the changed parts behave as intended and the unchanged parts are not adversely affected by the modifications. A typical way to perform regression testing is to keep a *regression test suite* (*i.e.*, a set of test cases) and to rerun all or part of it on the changed program. The effectiveness of regression testing, like the effectiveness of testing in general, highly depends on how well the regression test suite represents the way the program is used in the field. The problem of unrepresentativeness of a test suite can actually be more serious for regression testing than for testing of new applications: unlike the release of a new application, the release of a new version of an existing application is typically targeted to users who are already familiar with the application. Differences in behavior between the old and the new version are likely to generate more user dissatisfaction than, for instance, the lack of a feature in a new product.

Unfortunately, regression test suites often exercise the application in a very different way than the actual users, as shown by some of our previous work [12]. Our capture-replay technique could address this problem by generating regression subsystem and unit test cases from complete, real user executions captured in the field. These test cases would have the advantage of testing the software exactly in the way it is used on the field. These subsystem and unit test cases could be collected, for example, in the form of JUnit [5] test cases, and could be used to test new versions of such subsystems and units. Test cases could be captured in the field and sent back to the software producer, where they are collected and used as a regular regression test suite. Test cases could also be stored on the users' machine and be run on the new version of the program remotely, with only their outcome collected at the software producer's site. Like for the previous application, this second approach would allow for (1) eliminating privacy issues, and (2) reducing the amount of information to be transferred over the network and to be collected by the software developer.

4. THE TOOL: SCARPE

We implemented our technique in a tool called SCARPE (Selective Capture And Replay of Program Executions). SCARPE is written in Java and consists of three main modules:

- The *Instrumentation Module* adds probes to the program being captured or replayed. It instruments at the bytecode level, using the Byte Code Engineering Library (BCEL – <http://jakarta.apache.org/bcel/>).
- The *Capture Module* is invoked at runtime by the probes added by the instrumentation module. The module uses the information provided by the probes and suitably generates execution events.
- The *Replay Module* produces events for the observed code according to the execution events in the execution log. It also consumes events produced by the observed code.

Figures 3 and 4 provide a high-level view of how the tool works during capture and replay, respectively. As Figure 3 shows, there are two modes in which SCARPE can capture executions: on-line instrumentation and off-line instrumentation. In the *on-line instrumentation mode*, SCARPE runs the application being captured using a custom class loader. When a new class *c* is loaded, the class loader passes *c* to the instrumentation module, which checks whether *c* needs to be instrumented (*i.e.*, whether *c* belongs to the observed set or accesses fields of classes on the observed set). If so,

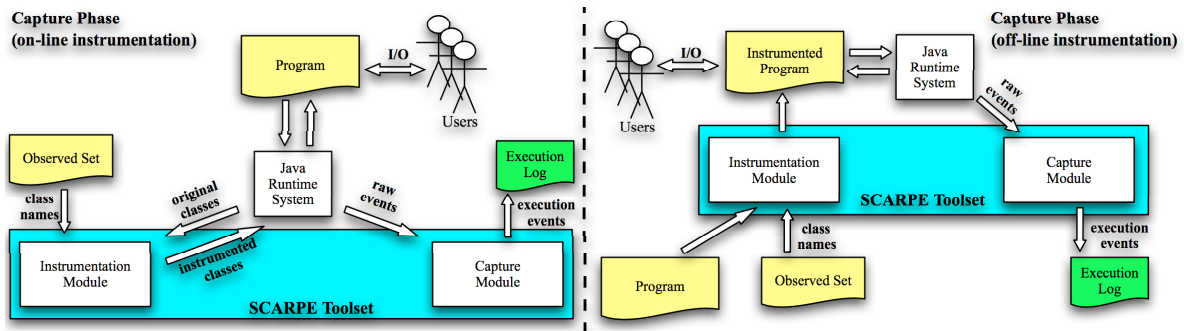


Figure 3: scarpe during capture.

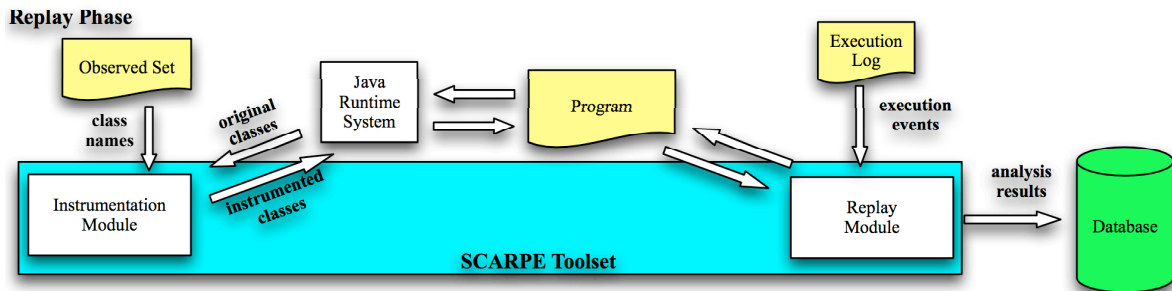


Figure 4: scarpe during replay.

the instrumentation module adds probes to the class and returns the instrumented class to the class loader. Otherwise, it simply returns the original class.

In the *off-line instrumentation mode*, SCARPE’s instrumentation module inputs the program to be captured and the observed set, instruments the classes in the observed set and the classes that access fields of such classes, and produces an instrumented version of the program that is ready to be deployed.

The decision of which instrumentation mode to use depends on the specific scenario in which SCARPE is being used. On-line instrumentation simplifies the capture phase because there is no need to save additional, instrumented versions of the program being captured or to use special classpaths. Moreover, on-line instrumentation adds flexibility to the capture process because it allows for capturing different subsets of the program by simply changing the file that contains the list of observed classes. However, the additional flexibility comes at a price. On-line instrumentation imposes an additional time overhead on the execution: the time required to instrument the classes on the fly. Therefore, especially when capturing users executions in the field, off-line instrumentation may be a better option than instrumentation on the fly.

In both modes, while users interact with the program, the probes added by the instrumentation module send *raw events* to the capture module. Raw events contain enough information for the module to build the actual *execution events* that are then recorded into an execution log.

During replay, as shown in Figure 4, the replay module inputs an execution log and suitably reproduces and consumes execution events, as described in Section 2.2. It is worth noting that SCARPE needs to instrument the program also during replay, to ensure that the program operates in a sandbox. To this end, all interactions between the observed

and the external code are transformed, through bytecode rewriting, into interactions between the observed code and the replay module. The replay module acts as the replay scaffolding, that is, it mimics the behavior of the part of the system that is not being replayed (*i.e.*, the external code). Because replay is performed without involving the user (and typically when free cycles are available), SCARPE always performs on-line instrumentation in this phase.

As discussed in Section 3, SCARPE can be used in several applications, such as regression testing, post-mortem dynamic analysis of users’ executions, and remote debugging. As an example, in Figure 4, we show the use of the tool for post-mortem dynamic analysis. In this case, the replay module could collect the results of the dynamic analysis and store them in a database, either locally or remotely, from which they can later be retrieved.

5. EMPIRICAL EVALUATION

To assess first the feasibility, and then the effectiveness and efficiency of our approach, we performed two empirical studies using SCARPE on two software subjects and an example. In the studies, we investigated two research questions:

- RQ1 – Feasibility: Can our technique correctly capture and replay different subsets of an application?
- RQ2 – Efficiency: Can our technique be used to capture executions in the field without imposing too much overhead on the fielded executions?

In the rest of this section, we present the studies that we performed to address our research questions.

5.1 Study 1: RQ1 – Feasibility

The goal of this study is to assess how reliable is our technique in capturing and replaying partial executions. To

achieve this goal, we selected a software subject and captured and replayed a large number of executions, using SCARPE for different subsets of the application. Although we recognize that this study can only provide limited guarantees of the correctness of our approach, we consider this evaluation an important first step towards more extensive assessments of the soundness of the approach.

As a subject for this study, we used NANOXML, an XML parser that consists of about 3,500 lines of code and 19 classes. We obtained NANOXML, along with a test suite consisting of 216 test cases, from the Subject Infrastructure Repository [23].

We performed the study in two parts. In the first part, we captured executions for subsystems containing only one class. For each class c in the application, we defined an observed set consisting of c only and ran all test cases in the test suite using SCARPE. In this way, we recorded 216 event logs (one for each test case in the test suite) for each of the 19 classes in the application, for a total of more than 4,000 logs. We then replayed, for each class, all of the recorded executions for that class.

In the second part of the study, we captured executions for observed sets of sizes two, three, four, and five. More precisely, we randomly created 25 observed sets for each of the sizes considered, so as to have a total number of 100 observed sets. Then, analogously to the first part of the study, we ran all of NANOXML’s test cases for each of the observed sets and later replayed the so collected execution logs. Therefore, overall, we recorded and replayed more than 20,000 partial executions (5,400 for each of the sizes considered).

In the first instance of the study, SCARPE failed for some of the executions involving observed sets with more than one class. Our investigation of the failures discovered that the problems were due to two faults in our implementation (and not to flaws in the technique). After we fixed these faults, no more failures occurred and the study resulted in a complete success: all executions were correctly captured and replayed. We checked the correctness of the replay both by making sure that all of the events generated by the observed set were matching the logged events and by spot checking some of the executions.

Although this is just a feasibility study, the successful capture and replay of about 25,000 executions is a promising result.

5.2 Study 2: RQ2 – Efficiency

The goal of Study 2 is to assess the efficiency of our approach. Several of the applications that we propose for our capture-replay technique involve the use of the technique in the field to capture real user executions. Therefore, it is very important that the approach can be used without imposing too much overhead on the fielded executions, which would make the technique impractical.

Although NANOXML is an appropriate subject for Study 1, because its size let us check and verify SCARPE’s results, for this second study we used a larger and more realistic subject: JABA. JABA (Java Architecture for Bytecode Analysis – <http://www.cc.gatech.edu/aristotle/Tools/jaba.html>) is a framework for analyzing Java bytecode that performs complex control-flow and data-flow analyses and consists of about 60,000 lines of code and 400 classes. JABA has an extensive regression test suite that was created and used over the last

several years of the system’s evolution. Because JABA is an analysis library, each test case consists of a driver that uses JABA to perform one or more analyses on an input program. The test suite that we used for the study contains 4 drivers and 100 input programs, for a total of 400 test cases.

To measure the efficiency of our technique, we proceeded as follows. *First*, we ran all 400 test cases and collected the normal execution time for each test case. SCARPE’s initialization code takes around 1.5 seconds to run, and this fixed cost does not depend on the specific test case or observed set considered. Therefore, test cases with an execution time of less than 1.5 seconds would necessarily show an overhead over 100%. To eliminate this problem, we discarded executions that took less than a second to run, which led to the elimination of 60 executions. (An overhead dominated by a fixed cost, when the cost is in the order of a second, would not provide meaningful information).

Second, for each driver, we identified nine classes in JABA that were covered by the test cases involving that driver. We considered a class covered by a test cases if the test case executed at least one of the class’s methods (the actual coverage was generally higher, and the called methods were often called many of times). We selected classes in this way to maximize the amount of data collected—capturing the execution of a class against a test case that does not covered it would result in an empty execution log.

Third, for each driver and each class c considered, we defined c as the observed set and ran all 340 (400 – 60) test cases using SCARPE. In this way we captured 3060 (340 * 9) executions of JABA test drivers.

Fourth, we measured the overhead imposed by SCARPE in terms of percentage increase between the execution times computed with and without SCARPE.

To run the tests, we used SCARPE in off-line instrumentation mode. In this way, we were able to compute the overhead due only to the collection of the events at runtime, without considering the overhead due to the on-line instrumentation (that can be avoided). Together with the timing data, we also collected the number of events captured for each execution, which is simply the number of entries in the execution log for that execution. We performed all experiments on a dedicated Pentium III, with 2GB of memory, running the GNU/Linux Operating System (2.6.16). We repeated all measures 10 times and averaged the results to limit the risk of imprecision introduced by external effects, such as caching.

Results and Discussion. Table 1 shows a summary of the overhead measured in the study. For each driver (ACDGDriver, CFGDriver, DefUseDriver, and ICFGDriver) the table shows the minimum, maximum, and average percentage overhead (computed over all of the test cases for that driver). The absolute minimum and maximum are highlighted using a boldface font.

Table 1: Summary overhead results for the four drivers.

	ACDGDrv	CFGDrv	DefUseDrv	ICFGDrv
<i>Min Ovh</i>	6%	3%	4%	4%
<i>Max Ovh</i>	494%	306%	877%	501%
<i>Avg Ovh</i>	104%	72%	145%	69%

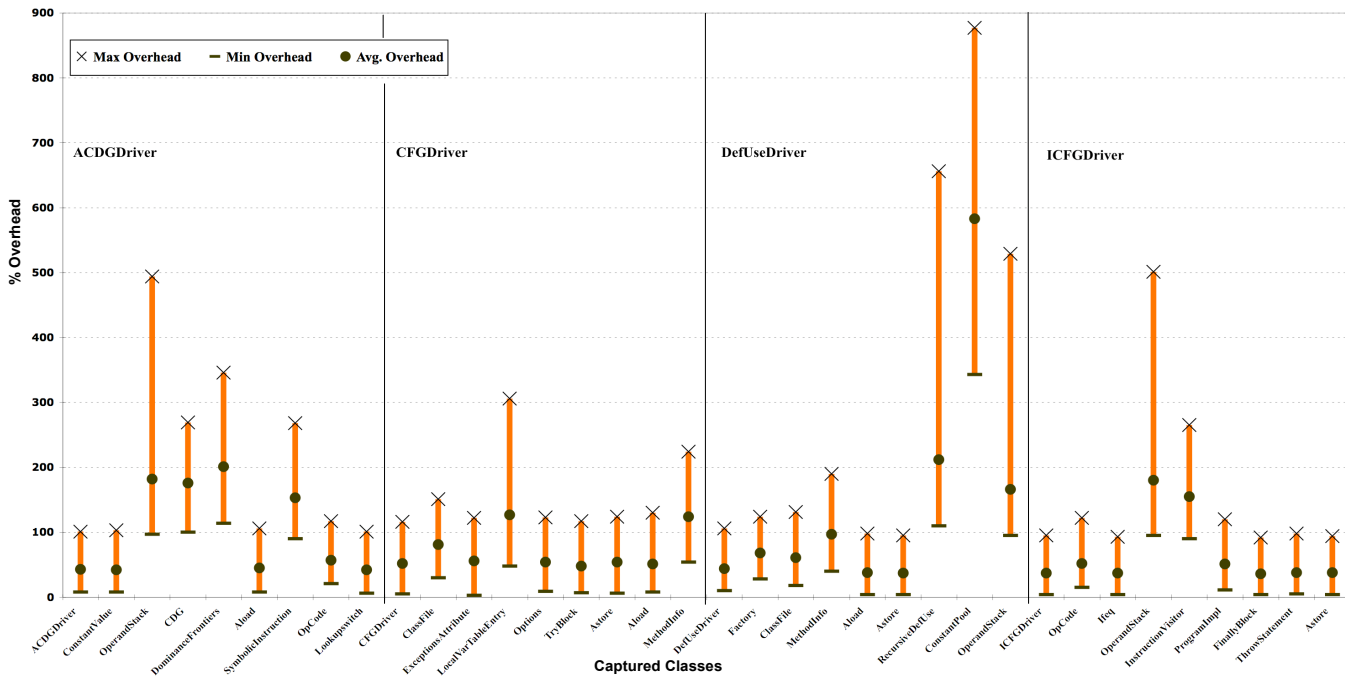


Figure 5: Performance results for scarpe.

To provide a better idea of the distribution of the results we also report, in Figure 5, the overhead information for each captured class (for each of the four drivers). Also in this case, the figure shows average, maximum, and minimum overhead (measured, this time, over the executions involving a specific captured class and driver). As far as space overhead is concerned, the average size of the execution logs collected, in uncompressed and unoptimized format, is in the order of 50KB per 1000 events (i.e., about 60MB for our largest log, consisting of 114,953,200 events).

As the results show, SCARPE’s overhead varies considerably across drivers and observed classes, ranging from 3% to 877%. A careful examination of the raw data collected in the study revealed that the cost of collecting a single event is similar across different types of events, and that the overhead is by and large proportional to the number of events collected per unit of time. For example, for driver ACDGDriver, the execution with 6% overhead generates 470 events, takes 22 seconds for the uninstrumented code, and takes about 23 seconds for the instrumented code. Considering the 1.5s fixed cost imposed by SCARPE, we can see that the overhead due to the 470 events is mostly negligible. For another example, for the same driver, the execution with the highest overhead (494%) produces 3,528,210 events, takes about 9 seconds during normal execution, and takes about 53 seconds when the code is instrumented. In this case, the high number of events per second is the cause of the considerable overhead. (The execution that causes 877% overhead has similar characteristics to this one). For yet another example, the lowest overhead for driver CFGDriver (3%) corresponds to an execution that produces 114,458 events, a fairly large number of events. However, in this case the execution takes about 87 seconds, that is, the number of events per second is two orders of magnitude lower than for the previous example.

Another observation that we can make, looking at the results shown in Figure 5, is that there are many cases in which the overhead imposed by SCARPE is on average between 30% and 50%, in the worst case around 100%, and in the best case in the single digits. Although 100% overhead is likely to be problematic in many cases, we must consider that our subject, JABA, is a processing intensive applications with no interaction with the user. There may be cases in which even a 100% overhead is not relevant for an application that runs completely in batch mode (e.g., overnight). More importantly, we hypothesize that the same overhead may become acceptable (or even negligible) for interactive applications, such as word processors or Web browsers. For these applications, the user “thinking time” is likely to decrease considerably the amount of events per second produced and also provide free cycles that SCARPE could leverage.

In fact, to get some initial evidence that could support our hypothesis, we performed an informal study in which both authors used a graph-drawing application while the execution of one of the drawing classes was being captured. Although we cannot claim any generality of the result, and our assessment is likely to be subjective, we can report that the slowdown in the drawing operations that involved the instrumented class was barely noticeable.

In summary, we believe that our results are encouraging and show that the approach we propose is feasible, especially considering that the optimization of SCARPE is still ongoing, and the performance of the tool can be further improved. Nevertheless, there are cases in which the current technique is likely to be too expensive to be used in the field. To account for these cases, we are currently investigating variation of the approach in which the capture stops after a given number of events or when the frequency of events produced (measured using a buffer that is flushed at regular intervals) is too high. We are also investigating

ways in which preliminary information, collected in-house, could be used to identify problematic part of an application (in terms of overhead) and exclude them from the capture.

6. RELATED WORK

The technique that is most related to ours is JRAPTURE, by Steven and colleagues [22], a technique and a tool for capture and replay of executions of Java programs. The technique replaces the standard Java API with a customized API that records all inputs to the running program, such as the data from a network socket or the keystrokes from the keyboard. During replay, another customized API feeds the recorded data back to the program. Although able to completely capture and replay executions, This technique incurs in many of the problems that we mention in the Introduction because it captures complete input/output information for each execution. Moreover, JRAPTURE requires two customized versions of the Java API for each Java version targeted.

The technique by Saff and Ernst for automated test factoring [20, 19] is also closely related to ours. Although the two techniques share some commonalities in terms of technical approach, they have different goals and, partly because of this, significant technical differences. Their technique aims to improve the efficiency of testing by automatically building mock objects for testing units in isolation, and is designed to be used in-house. Our main goal is to capture partial user executions for deployed software for later replay. Because their technique is not constrained by the requirement of being usable in the field, it has different trade-offs than ours, in particular in terms of complexity of the deployment and overhead. For each class, their technique creates an interface, modifies the class so that it implements this interface, and replaces references to the class with references to the interface. A capturing class is then introduced that records the interaction between internal and external code and delegates the actual work to the original class. The Java API classes are also massively instrumented in the same way, off-line. In our technique, we only need to instrument Java API classes that do not have a default constructor, to add one. Moreover, this is just an implementation shortcut that could be avoided. Also, their technique requires to replace all field accesses with corresponding calls to getter and setter methods, which can impose a considerable time overhead. Finally, Saff and Ernst do not discuss how some aspects of the Java language are handled (*e.g.*, exceptions) and do not report overhead data, so we cannot compare the two techniques along these dimensions. Overall, we feel that the two techniques, although related, are mostly complementary.

Other related techniques aim to reproduce the concurrent behavior of applications (*e.g.*, [1, 8, 18]). In particular, Choi and colleagues present DejaVu, a platform for analyzing multi-threaded Java program executions [1]. DejaVu supports debugging activities by performing a fully-deterministic replay of non-deterministic executions and allowing for inspecting the state of the program at any point of the execution. DejaVu and our technique also have different goals and, thus, a different set of constraints. DejaVu focuses on reproducing, giving the same inputs, the same application behavior in terms of concurrency-related events, has no need to store input and output values, and does not have efficiency constraints (being targeted to debugging). As discussed above, our technique is mostly concerned with

automatically capturing and replaying subsystems and has efficiency as a priority.

A general difference between our technique and most other existing techniques is that most existing techniques work at the runtime-system level, by either modifying the Java Virtual Machine or by using a customized version of the system libraries, whereas our approach works at the application level, through bytecode rewriting. Because our technique does not require any runtime-system support, it can be used on any platform that provides a standard implementation of the Java Virtual Machine, which improves portability and enables the use of the technique on users' platforms.

7. CONCLUSION

In this paper, we have described a technique for partial capture and replay of user executions. The technique allows for efficiently capturing executions for a given subsystem and replaying such executions on the subsystem in a sandbox. We have also discussed three possible applications of the technique and presented a tool, SCARPE that implements the technique and that is currently available for download (<http://www.cc.gatech.edu/~orso/software.html>). Finally, we have presented an empirical evaluation of the technique that shows that, although there is room for improvement, the approach is feasible, and capture-replay of executions can be performed efficiently in many cases.

Our immediate goal in future work is to improve the implementation of SCARPE in terms of performance, to allow it to run efficiently also for executions that generate a large number of events. This improvement will be performed in parallel with additional experimentation on more subjects and executions, to further refine and extend the approach.

Other future work involves the investigation of the three applications proposed in the paper: post-mortem dynamic analysis of users' executions, debugging of deployed applications, and regression testing based on users executions. We are currently working on the first and second applications. We are investigating the use of post-mortem dynamic analysis to collect expensive coverage measures, such as data-flow coverage, in the field. We are also developing a tool that combines SCARPE and delta debugging [25].

Finally, another potential direction for future work is the use of static analysis and, possibly, profiling information to help users decide which classes should be included in the observed set. Currently, the technique does not provide any support for this task. Static analysis and profiling could help identify classes that are tightly coupled and that, if separated, may generate an impractical number of interactions. Appropriate analyses could suggest the users ways to improve the performance of the technique by either including more classes in the observed set (*e.g.*, classes that are tightly coupled with classes already in the set) or by excluding some classes (*e.g.*, classes tightly related with external code and not cohesive with the observed set).

Acknowledgments

This work was supported in part by NSF awards CCR-0205422, CCR-0306372, and CCR-0209322 to Georgia Tech. Bryan Kennedy helped implementing a first prototype of SCARPE. Gregg Rothermel provided the software subject and the test cases for our empirical evaluation.

8. REFERENCES

- [1] B. Alpern, T. Ngo, J.-D. Choi, and M. Sridharan. Dejavu: Deterministic java replay debugger for jalapeño java virtual machine. In *Proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000) – Addendum*, pages 165–166, 2000.
- [2] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2002)*, pages 2–8, Charleston, SC, USA, november 2002.
- [3] S. Elbaum and M. Diep. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.
- [4] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 451–462, November 2004.
- [5] E. Gamma and K. Beck. JUNIT: A Regression Testing Framework. <http://www.junit.org/>, 2005.
- [6] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying Classification Techniques to Remotely-Collected Program Execution Data. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 146–155, Lisbon, Portugal, september 2005.
- [7] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, Dec 2000.
- [8] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *In Proceedings of the 14th IEEE International Parallel & Distributed Processing Symposium*, pages 219–228, 2000.
- [9] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)*, June 2005.
- [10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Addison-Wesley Pub Co, 1999.
- [11] Microsoft online crash analysis, 2004. <http://oca.microsoft.com>.
- [12] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European Software Engineering Conference and 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, pages 128–137, Helsinki, Finland, september 2003.
- [13] A. Orso and B. Kennedy. Improving dynamic analysis through partial replay of users' executions. In J. Choi, B. Ryder, and A. Zeller, editors, *Online Proceedings of the Dagstuhl Seminar on Understanding Program Dynamics*, Dagstuhl, Germany, december 2003. <http://www.dagstuhl.de/03491/Proceedings>.
- [14] A. Orso and B. 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.
- [15] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 241–252, Newport Beach, CA, USA, november 2004.
- [16] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Softw. Eng., 1999*, pages 277–284, May 1999.
- [17] G. Rothermel and M. J. Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.
- [18] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of ACM SIGPLAN Conference on Programming Languages and Implementation (PLDI)*, pages 258–266, 1996.
- [19] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic Test Factoring for Java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE 2005)*, pages 114–123, November 2005.
- [20] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 49–51, Washington, DC, USA, June 7–8, 2004.
- [21] J. Seward. Valgrind, an Open-Source Memory Debugger for x86-Gnu/Linux, 2002. <http://valgrind.kde.org/>.
- [22] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, pages 158–167, 2000.
- [23] Subject Infrastructure Repository. <http://cse.unl.edu/~galileo/php/sir/index.php>, 2006. University of Nebraska-Lincoln.
- [24] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 45–54, 2004.
- [25] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 28(2):183–200, 2002.