Selective Capture and Replay of Program Executions

Alessandro Orso and Bryan Kennedy
College of Computing
Georgia Institute of Technology
orso@cc.gatech.edu, bck@acm.org

ABSTRACT

In this paper, we present a technique for selective capture and replay of program executions. Given an application, the technique allows for (1) selecting a subsystem of interest, (2) capturing at runtime all the interactions between such subsystem and the rest of the application, and (3) replaying the recorded interactions on the subsystem in isolation. The technique can be used in several scenarios. For example, it can be used to generate test cases from users' executions, by capturing and collecting partial executions in the field. For another example, it can be used to perform expensive dynamic analyses off-line. For yet another example, it can be used to extract subsystem or unit tests from system tests. Our technique is designed to be efficient, in that we only capture information that is relevant to the considered execution. To this end, we disregard all data that, although flowing through the boundary of the subsystem of interest, do not affect the execution. In the paper, we also present a preliminary evaluation of the technique performed using SCARPE, a prototype tool that implements our approach.

1. INTRODUCTION

The initial motivation for this work stems from results that we obtained while experimenting with a technique that leverages field data to perform impact analysis and regression testing [5]. These results clearly showed that, for the cases considered, executions in the field manifest a quite different behavior than in-house executions. Therefore, we wanted to leverage such differences to improve in-house testing. Ideally, we wanted to be able to capture executions in the field and then replay them in-house.

More generally, the possibility of capturing and replaying program executions can be useful for many software-engineering tasks. In testing, for instance, the ability to capture and replay executions would 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. 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 uninstrumented program and then perform the expensive analyses off-line, while replaying.

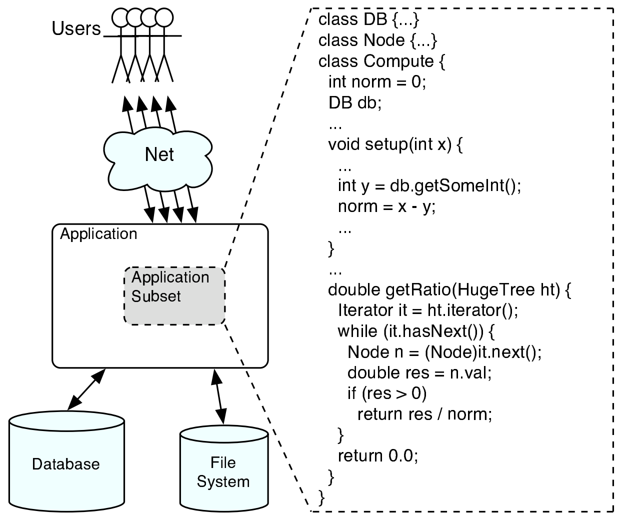
Workshop on Dynamic Analysis (WODA 2005) 17 May 2005, St. Louis, MO, USA
Copyright 2005 ACM ISBN # 1-59593-126-0 ...$5.00.

Unfortunately, capturing complete executions is generally infeasible, for several reasons. First, there are practicality issues. To capture a complete execution, we may need to record a large volume of data—all the inputs to the application. Also, capturing the inputs provided to an application can be difficult and may require custom mechanisms, depending on the way the application interacts with its environment. Second, there are privacy issues. The data captured could contain confidential information that users may not want to be collected. Third, there are issues related to side effects. If a captured execution has side effects on the system on which it runs, replaying it may corrupt the system. Furthermore, the environment may have changed between capture and replay time.

To address these problems, while still being able to reproduce executions, we defined a novel technique based on *selective capture and replay of executions*. Given an application, the technique lets us (1) select a subsystem of interest, (2) capture at runtime all the interactions between such subsystem and the rest of the application, and (3) replay the recorded interactions on the subsystem in isolation. Our technique is designed to be efficient: for each execution, we only capture information that is relevant to that execution. To this end, we disregard all data that, although flowing through the boundary of the subsystem of interest, do not affect its execution. Intuitively, our technique captures only the minimal subset of the application's state and environment required to replay the execution considered on the selected subsystem.

Our technique allows for addressing the issues of practicality, privacy, and safety listed above. When practicality is concerned, we can limit the volume of data that we need to record by suitably selecting the subset of the application for which we capture information. Also, we address the problems represented by complex execution environments because we always capture (and replay) at the boundary between parts of the application. When privacy is concerned, we can exclude from the subsystem of interest those parts of the application that handle confidential information. When this is not possible, we envision a use of our technique in which also the replay is performed on the users' machines. For example, if the technique is used to perform expensive dynamic analyses on part of the application, we 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. 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.

An additional advantage of our technique is that, by performing capture and replay at the subsystem level, it enables additional applications that would not be possible for a technique that captures complete executions. In particular, our technique can be used to



```
class DB {...}
class Node {...}
class Compute {
  int norm = 0;
  DB db;
  ...
  void setup(int x) {
    ...
    int y = db.getSomeInt();
    norm = x - y;
    ...
  }
  ...
  double getRatio(HugeTree ht) {
    Iterator it = ht.iterator();
    while (it.hasNext()) {
      Node n = (Node)it.next();
      double res = n.val;
      if (res > 0)
        return res / norm;
    }
    return 0.0;
  }
}
```

Figure 1: Example application.

automatically generate subsystem and unit test cases from system test cases (or from complete executions in general). Because of the way we capture and replay, such test cases would include all needed stubs, drivers, and even oracles, and could benefit testing activities such as regression testing.

In this paper, we present our technique for selective capture and replay. We also present a prototype tool, called SCARPE, that implements our technique for Java programs, and a study that shows the feasibility of the approach. Finally, we discuss several direction for future research.

2. SELECTIVE CAPTURE AND REPLAY

2.1 Overview

Before presenting our technique for selective capture and replay, we provide an example that we use in the rest of the discussion to motivate and illustrate the technique. Figure 1 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 the information required to replay the entire application would involve technical challenges (e.g., collecting the data that flow from the users to the application and vice versa), storage problems (e.g., we may have to record consistent portions of the database), and privacy issues (e.g., the information provided by the users may be confidential). We use this kind of application as an example because it lets us stress 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).

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, it 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*.

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

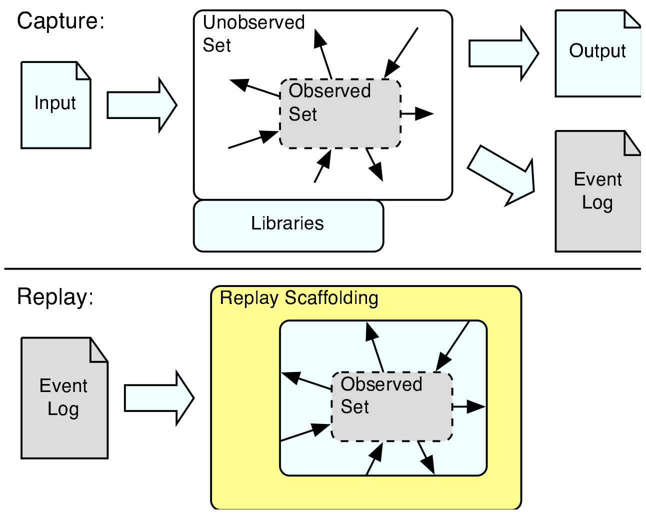


Figure 2: Overview of the technique.

technique reads the recorded set of events and replays the corresponding interactions.

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.

2.2 The Technique

Because the characteristics of the programming language targeted by the technique considerably affect its definition, we define our technique for a specific language: Java.[1] 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 or a subset thereof.

In the rest of the paper, we use the following terminology. 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`, or classes containing native methods), 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.

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. 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

[1]We refer to Java before the introduction of generic classes.

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. 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. 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

A major issue, when capturing data flowing through the boundary of a subsystem (e.g., values assigned to a field) is that 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 time overhead of over 500% 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.

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 (*incall*s) and calls from the observed to the unobserved code (*outcall*s). 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: (1) *OUTCALL* events, for calls from observed to unobserved code; (2) *INCALL* events, for calls from unobserved to observed code; (3) *OUTCALLRET* events, for returns from outcalls; and (4) *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.

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: (1) *OUTREAD* events, for read accesses from observed code to unobserved or library fields; (2) *OUTWRITE* events, for write accesses from observed code to unobserved or library fields; and (3) *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 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: (1) *EXCIN*, for exceptions that propagate from external to observed code; and (2) *EXCOUT*, 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 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.

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.

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, we simply hard-coded the constructor to be used in these special cases, 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.

Note that, whereas recording INCALL, INWRITE, OUTCALLRET, and EXCIN events is necessary to replay executions, the need for recording the events generated in the observed code 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, events originated in the observed code 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 those 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. The retrieved object can be `null`, an actual object, if its class is part of the observed set, or a (possibly newly-created) placeholder 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. The 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`):[2]

```
Object tmp = scaffolding.consumeCall(''foo/HugeTree'',
                             < object ID for ht >,
                             ''iterator:()Lfoo/Iterator'',
                             < empty array of paramters >);
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 this is not the case, 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.

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'',
                                     < objectIDforn >,
                                     ''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

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

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.

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 not, an error is reported to the user.

2.5 Additional Considerations

For space reasons, we glossed over several technical details. In this section, we concisely discuss the most relevant ones.

Assumptions: Our technique works under some assumptions. 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 classes in the application. We also assume that the interleaving due to multithreading 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.

Special handling of specific language features: 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), additional instrumentation is required. Analogously, to correctly handle all accesses to arrays, some additional instrumentation is required. Finally, inheritance and access modifiers require some special handling. In particular, 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).

3. EMPIRICAL EVALUATION

To evaluate our technique, we built SCARPE (Selective Capture And Replay of Program Executions), a prototype tool that implements our technique, and used it on a software subject. During capture, SCARPE runs the application being captured using a custom class loader. Such class loader inputs the list of observed classes and instruments (modifiable) classes on the fly, at class-loading time, using the Byte Code Engineering Library (BCEL).[3] In this way, we simplify the capture phase because there is no need to save instrumented versions of the application and related libraries and to use special class paths during execution. This aspect is especially important if the technique is to be used on users' machines.

During replay, SCARPE acts as the replay scaffolding. It mimics the behavior of the part of the system that is not being replayed (i.e., the external code) and suitably reproduces and consumes events. Also in this case, the necessary instrumentation is performed on the fly, by means of another custom class loader.

[3]`http://jakarta.apache.org/bcel/`

Using SCARPE, we performed a feasibility study. The goal of the study was to assess whether the technique can be used on something more than a toy application and to evaluate its practicality. As a subject, we used NANOXML, an XML parser that consists of 19 classes and about 3,300 lines of code. To execute NANOXML, we used a test suite, developed by other researchers, that contains 216 test cases. 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 the recorded executions for that class.

The feasibility study was a complete success, in that 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. (To this end, we created a special version of SCARPE that reports all of the events generated by the observed code and lets the user inspect them.)

Although this is just a feasibility study, and the evaluation of our technique is still in its early stages, we consider the successful capture and replay of thousands of executions a very promising result.

4. RELATED WORK

Several techniques have been defined for capture and replay of entire applications. The technique that is most related to ours is JRAPTURE, by Steven and colleagues [9], 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. During replay, another customized API feeds the recorded data back to the program. 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.

Other related techniques aim to reproduce the concurrent behavior of applications (e.g., [1, 2, 3, 7, 10]). In particular, Choi and colleagues present DejaVu, a platform for analyzing multi-threaded Java program executions [1, 2]. DejaVu supports debugging activities by performing a fully-deterministic replay of non-deterministic executions. DejaVu and our technique have different goals and, thus, a different set of constraints and tradeoffs. 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. Our technique is mostly concerned with automatically capturing and replaying subsystems and has efficiency as a priority because we want the technique to be usable also on deployed software.

Recently, Saff and Ernst presented a technique for automated test factoring that aims at improving the efficiency of testing by automatically building mock objects for testing units in isolation [8]. Although interesting, their approach is still preliminary. Our technique seems to provide better support for Java's object-oriented features and characteristics (e.g., it is not clear how their approach would handle exceptions and unmodifiable classes). Also, our approach is more general and can be used for different applications. In fact, our replay scaffolding could be used as a set of mock objects for the subsystem or unit of interest.

5. CONCLUSION AND FUTURE WORK

We presented a novel technique for selective capture and replay of program executions, a tool that implements the technique, and a study performed on a real application that shows the feasibility of the approach. There are many possible directions for future work.

In the immediate, we will continue our evaluation of the approach to assess the performance of our technique on various subjects and executions. In particular, we must evaluate our technique in the case of multi-threaded programs. To this end, we have started collecting subjects and improving our tool. Through experimentation, we will assess the efficiency of our technique and its limits. The results will drive refinements or extensions of our approach.

Other research directions consist of investigating the use of the technique for various applications. We discuss a few possibilities.

A first possible application is *post-mortem dynamic analysis of users' executions*. Our technique could be used to selectively capture users executions and to perform dynamic analysis (e.g., coverage or performance analysis) of the observed code while replaying these executions. We envision the investigation of this application in several scenarios. One scenario involves the collection of the users' executions to replay them in-house. Another possibility is to replay and analyze the executions on the users' machines, leveraging free cycles, and collect the results of the analysis only. (Note that this second scenario would eliminate most privacy issues: the only data collected from the users would be analysis results, such as performance data, possibly further sanitized.) Yet another possibility is to use some criterion for deciding which executions to delete and which ones to gather. For example, only executions that terminate with an exception could be sent back for analysis.

Another possible application is *regression testing*. Subsystem and unit test cases could be generated from complete executions of the application (e.g., as JUnit test cases) and used to test new versions of such subsystems and units. The major issue with this application is that our technique collects minimal information during capture. The technique may not be able to replay the captured execution when the interactions between observed and external code change in the new versions. For example, consider a new version of class `Compute` (see Figure 1) in which method `getRatio` accesses field `size` of object `ht`. In such a case, we would not be able to replay the execution of `Compute` because our log would contain no information about an access that was not occurring in the original version of the class. For this application of the technique, it is first necessary to study sets of changes between versions of various programs to assess how often the replay on a different version of the observed set would fail. One way to do this is to capture executions for a given version of a class or subsystem, replay them on a set of subsequent versions of such class or subsystem, and measure how often the replay results in *out-of-sync* events. In some cases, it may be possible to simply ignore such out-of-sync events (e.g., when the event does not affect the main flow of the computation). In other cases, those events may be handled by providing some default value to the observed code (e.g., for OUTCALLRET events that corresponds to unmatched OUTCALL events). We will also investigate ways to extend the amount of information captured and to balance the resulting trade-offs between efficiency and effectiveness. For example, we are already considering the possibility of capturing complete objects of some classes, such as `String`.

A third application is *debugging*. Consider again the example in Figure 1. The example contains a 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. In this context, we could selectively capture the execution of a subsystem of interest (e.g., the one in which the failure occurs) and then debug its replayed execution. In this context, we are currently working on combining our approach with Zeller's delta debugging [11], to find a minimal set of interactions that lead to the failure.

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 requires the users to define the observed set and 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. (Although our technique collects minimal information, there may still be cases in which the execution log becomes too large to be practical.) 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). Static analyses such as flow analysis could also be used to identify classes that should be excluded from the observed set because they handle confidential data.

Finally, from a more practical standpoint, a possible research direction is to modify the technique to work at the Java Virtual Machine, rather than at the bytecode, level. While this approach sacrifices portability, it has the potential to improve the performance of the capture phase, as well as to allow for a more accurate replay of threaded programs [2].

Acknowledgments

This work was supported in part by NSF awards CCR-0205422, CCR-0306372, and CCR-0209322 to Georgia Tech. Shrinivas Joshi helped implementing the tool. The attendants of the Dagstuhl Seminar on "Understanding Program Dynamics" gave useful comments on an early version of this work [6]. Gregg Rothermel provided the software subject and the test cases for our empirical evaluation.

6. 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 OOPSLA 2000 – Addendum*, pages 165–166, 2000.

[2] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[3] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *Proceedings of the Intl. Parallel & Distributed Processing Symposium*, pages 219–228, 2000.

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Addison-Wesley Pub Co, 1999.

[5] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of ESEC/FSE 2003*, pages 128–137, september 2003.

[6] 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*, December 2003. `http://www.dagstuhl.de/03491/Proceedings`.

[7] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of PLDI 1996*, pages 258–266, 1996.

[8] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In *Proceedings of PASTE 2004*, pages 49–51, June 2004.

[9] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of ISSTA 2000*, pages 158–167, 2000.

[10] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE TSE*, 17(1):45–63, 1991.

[11] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TSE*, 28(2):183–200, 2002.