Automatic Extraction of Object-Oriented Component Interfaces

John Whaley Michael C. Martin Monica S. Lam

Computer Systems Laboratory
Stanford University
{jwhaley, mcmartin, lam}@stanford.edu

ABSTRACT

Component-based software design is a popular and effective approach to designing large systems. While components typically have well-defined interfaces, sequencing information—which calls must come in which order—is often not formally specified.

This paper proposes using multiple finite state machine (FSM) submodels to model the interface of a class. A submodel includes a subset of methods that, for example, implement a Java interface, or access some particular field. Each state-modifying method is represented as a state in the FSM, and transitions of the FSMs represent allowable pairs of consecutive methods. In addition, state-preserving methods are constrained to execute only under certain states.

We have designed and implemented a system that includes static analyses to deduce illegal call sequences in a program, dynamic instrumentation techniques to extract models from execution runs, and a dynamic model checker that ensures that the code conforms to the model. Extracted models can serve as documentation; they can serve as constraints to be enforced by a static checker; they can be studied directly by developers to determine if the program is exhibiting unexpected behavior; or they can be used to determine the completeness of a test suite.

Our system has been run on several large code bases, including the joeq virtual machine, the basic Java libraries, and the Java 2 Enterprise Edition library code. Our experience suggests that this approach yields useful information.

1. INTRODUCTION

A popular approach to designing large systems is component-based software design. The idea is to improve software reuse by crafting carefully engineered software elements suitable for a broad array of applications. Also, once the applications programming interface (API) has been designed, implementors can lower the number of cross-application dependencies that their code needs to worry about, concentrating instead on providing the services that the API specifies. As a result, modern enterprise computing architectures are not monolithic systems, but typically consist of multiple tiers of reusable components such as user interfaces or system services such as transactions, security, and databases. The resulting component libraries can then be reused across many different applications.

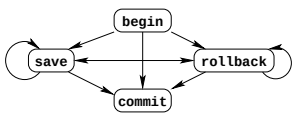

Figure 1: FSM for the database example

The API of these components often includes constraints on when a public method may be invoked. For example, an SQL server may include the commands `begin`, `commit`, `save`, and `rollback`. The command `begin` must first be executed, then a series of `save` and `rollback` commands can be issued, and then, finally, `commit`. It would be incorrect, for example, if a `save`, `rollback`, or `commit` is invoked before any `begin` command. We can represent these constraints as a finite state machine (FSM), as shown in Figure 1. Constraints such as these are very common in APIs.

There has been a good deal of recent work that deals with modeling objects with finite state machines. Previous systems like Vault[7], NIL and Hermes[23] provide programmers with linguistic constructs to specify the state of the variables in a program, and a compiler is used to ensure that the object is always in the correct state. The advantage of this approach is that code written in this way is guaranteed to conform to the finite state models. This, however, requires programmers to rewrite their code; retrofitting large systems of existing code into this framework is not plausible.

Systems such as PREfix[3], Metal[5, 8] and SLAM[2] operate directly on existing software; they check if the code conforms to pre-defined correctness constraints, many of which can be expressed as FSMs. These systems have been demonstrated to be successful in finding many errors in operating systems.

One of the bottlenecks of such systems is that the correctness constraints are often not readily available for large systems. Few programmers are both willing and able to

This research was supported in part by NSF award #0086160, an NSF student fellowship, and a Stanford Graduate Fellowship.

write down the specifications. Furthermore, programmers may also be wrong when specifying the constraints. To deal with this, there have been various recent attempts to infer constraints automatically. Engler et al. [16] proposed techniques to infer constraints from programs by analyzing the behavior of the code statically. To allow for errors in the program, behavior that is observed most of the time is considered the norm. Their results suggest that this approach is effective in finding bugs like whether a lock function call is followed by an unlock. In addition, dynamic techniques to extract information directly from programs have also been proposed[1, 9]. In particular, Ammons et al. use machine learning techniques, with limited success, to extract specifications from the program by analyzing dynamic program traces[1].

1.1 The Problem

The focus of this paper is on developing techniques that can automatically extract application interfaces directly from existing code. However, unlike most of the previous work in this area, we are interested in techniques that apply to large object-oriented component-level software rather than system code written in low-level languages like C. Large applications typically manipulate many dynamically allocated objects stored in recursive data structures. Instances of the same class may all follow the same state transition model, but they can be in different states at any one time. The states they are in may be encoded in some cases by explicit variables; however, more often than not, their states are governed by highly application-specific semantics that are unknown to the programming tools. Techniques that track the control flow or values of variables through limited programming paths are not effective here. Because of our interest in handling large software, on the order of millions of lines of code, formal methods such as model checking are also not feasible.

Our techniques are designed to take advantage of the modularity in object-oriented programming languages. In a well-designed object-oriented system, the set of public methods in each class definition defines a complete interface for all instances of objects belonging to the class. It is also guaranteed that only the methods of a class can mutate the state of the object. We can thus operate at the method level: instead of dealing with the definitions and uses of individual memory locations, we can focus on finding higher-level relations like how methods such as `commit` and `rollback` relate to each other.

This paper proposes using multiple FSM submodels to model the interface of a class. A submodel includes a subset of methods that, for example, implement a Java interface, or access some particular field. Each state-modifying method is represented as a state in the FSM, and transitions of the FSMs represent allowable pairs of consecutive methods. In addition, state-preserving methods are constrained to execute only under certain states. We explore two fully automatic techniques for extracting partial interface models from large Java programs, one being a dynamic tool and the other a static analysis tool.

Our dynamic analysis technique observes the program as it runs to build up a model that reflects the operations performed during the execution of the program. The dynamic analysis does not suffer the disadvantage of static analyses, which must necessarily be conservative. Here, the results gathered are based on complete sequences of events that can actually take place. Unfortunately, a dynamic system can only prove things existentially. Any purported universal that a dynamic system finds may prove to only be an artifact of the test cases that the system was trained on.

To demonstrate that a model actively forbids certain cases, we must turn to static analysis. Our static analysis technique searches for method sequences on a component that invariably cause exceptions to be thrown. Such sequences are deemed illegal. A static system by itself, however, suffers from the disadvantage that it cannot easily identify a priori whether or not a given sequence is part of typical or permitted usage. The advantages and disadvantages of the static and dynamic approaches complement one another nicely; a hybrid system that uses them both can acquire a great deal more information than either one alone.

The hybrid system can also take advantage of further advantages of the dynamic and static approaches that do not directly pertain to the model. A static system has fuller access to the structure of the code, and can make a variety of deductions about the nature of the classes to let the dynamic system focus more carefully on useful features of the program runs. The dynamic system in turn has easy access to the current state of the heap at any given time, so there is no need for advanced pointer alias analysis on the static side. Deductions which require detailed alias information can simply be delayed until run time.

1.2 Applications of Automatically Extracted Models

Partial models extracted by our system are useful in several ways. First, the information can serve as documentation of the system. This is especially useful for programmers faced with using a large piece of software for the first time. Second, component designers may wish to examine the extracted model to see if it coincides with their expectations. Third, the dynamic analysis can be used as a means to measure the coverage of a test suite. The model extracted from analyzing the test suite gives a succinct summary of the sequencing of methods tested. These results may suggest additional tests be written to test important subsequences of method invocations. Fourth, given a model that may have been either extracted automatically or prepared by the programmer, we can monitor the execution of a program and automatically flag errors that deviate from correct behavior. Finally, static tools to check the conformance of the model can be implemented to locate errors in programs. We have implemented these analyses and systems to operate on Java bytecodes.

In this paper, we present several case studies to show that our relatively simple but scalable techniques are successful in extracting useful models from several large programs. We tested our tools on four different applications that together consist of over 1.2 million lines of code. We demonstrate:

1. *The effectiveness of the model.* The model automatically extracted for the socket implementation in `java.net` illustrates how our interface models are effective in capturing the correct usage of a class.

2. *Automatic static model extraction.* Our static analysis successfully extracts useful models for the standard Java libraries. The information is useful as documentation; also, they can be used by static tools to look

for errors in programs.

3. *Use of the model to characterize test suites.* We demonstrate scalability by using the dynamic analysis tool on the J2EE enterprise edition platform. We show that the extracted models convey interesting information about the program being tested and the test suite itself.

4. *Use of the models in software auditing.* To get a sense of the applicability of our system for program evolution and developer feedback, we applied our technique to a program that we are familiar with: the joeq virtual machine. The dynamic system was able to find some discrepancies between the intended and implemented API, and the static system provided an accurate rendition of the appropriate call sequence.

1.3 Paper Overview

We first define our model of component interfaces in Section 2, illustrate it with an example, and discuss how to optimize its utility. We describe our static analysis in Section 3 and our dynamic analysis in Section 4. We present our preliminary experience in Section 5. Section 6 discusses related work and Section 7 concludes.

2. A COMPONENT INTERFACE MODEL

The design of our model of the interface is inspired by the concept of *path expressions*[4]. Path expressions were originally designed to allow simple specification of complex synchronization schemes for concurrent processes and operating systems. Regular expressions are used to define the set of admissible execution histories of operations on a shared resource. These histories represent an interleaving of actions on multiple threads.

Our approach is based on a simplified version of path expressions that treat objects as a resource shared between different sections of an application. Instead of allowing a general regular expression to capture the history of methods invoked on an object, we instead place restrictions on our FSMs to make them easier to extract and more efficient to enforce dynamically.

A naïve approach is to have every method in the model represented by exactly one state.

Definition 2.1 *Naïve model.* A naïve interface model of a class cl is a finite state machine, denoted $M_{cl} = \langle S, T \rangle$, where S is the set of methods in class cl plus the two special methods START and END, and $T \subseteq (S \times S)$ is the set of legal method sequences. An instance is in state op if op is the last method from the set S that has been invoked on the instance, or in state START if no such method exists. A method $op' \in S$ can be invoked on an instance if and only if the instance is in state op and $(op, op') \in T$.

There are two major problems with this naïve model. First, most objects have sufficiently complex sequencing constraints that merely knowing the last method called cannot capture the proper behavior of the object. Secondly, many methods are *state-preserving*: they do not have any side effects and do not advance the state of the object. Including these in the naïve model destroys its accuracy.

In the remainder of this section we refine our naïve model to address these issues. Section 2.1 discusses the issue of more complex sequencing constraints, and Section 2.2 discusses state-preserving transitions. Section 2.3 concludes with a formal specification of the model.

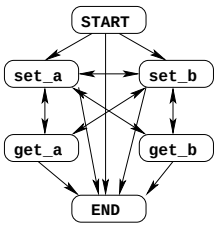

Figure 2: Naïve model example

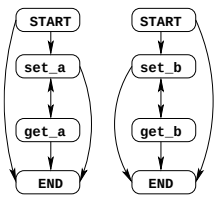

Figure 3: Improved model with field splitting

2.1 Model Slicing by Type and by Member Fields

Models are associated with individual classes; whether the class is an instantiable object or some abstract superclass is unimportant to the model. Most objects are members of multiple classes—they are members of a class that subclasses some other type, or they implement a number of interfaces. Each type, with its methods, encapsulate a set of constraints. Thus, each type to which an object may be cast specifies a separate model, which in turn is represented as an FSM. The full set of sequencing constraints for the object are represented by the product of the FSMs.

By the same token, we also slice the methods in a class by their field accesses. More precisely, given a class of n fields, we create up to n submodels, where each submodel contains the methods (not necessarily unique to that submodel) that access the same field. A submodel may subsume another if its methods and allowable method pairs are a superset of the other. The rationale for slicing by fields is that unless methods refer to the same variable, they are not related, and their relative ordering is independent of each other. Representing them separately allows us to accurately model independent aspects of a single object.

Consider a simple class with two fields `a` and `b`, and four methods, `set_a`, `get_a`, `set_b`, and `get_b`. Here, the method `get_a` cannot be invoked before `set_a`, and similarly, the method `get_b` cannot be invoked before `set_b`. Attempting to model this behavior with the naïve model produces the model shown in Figure 2. Notice that in order to allow arbitrary interleaving of `get` and `set` operations, we had to give up the capability of enforcing that both `a` and `b` be defined before use. If we split the model in two, we get the FSMs shown in Figure 3. This new system is thus able to enforce that `get_a` and `get_b` are preceded by `set_a` and `set_b`, respectively.

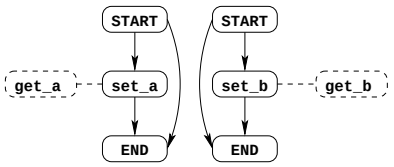

Figure 4: Final model for get/set example

2.2 State-Preserving Transitions

In a naïve model, every method call changes the state of the model. However, not all methods update the state of the object; the `toString` and `hashCode` methods defined on all Java objects are examples of this. Having methods like these in our model allows any method to be followed by any other method, provided the state-preserving method is invoked in between. A common approach used to improve accuracy is to create finite state machines that keep track of the last k calls instead of only the most recent call[21]. However, using such an approach would only exacerbate the problem. The model may now contain up to n^{2k} transitions, greatly complicating the analysis. Furthermore, it is legal to make k state-preserving calls in a row, again landing the object in a state that can reach any other state.

Our approach is to distinguish between methods that produce side effects and those that do not. Side-effect free methods do not advance the state of the object; common examples of side-effect free methods in Java include `toString` and `hashCode`. Including them would create an overly weak model as discussed above. We thus ignore these side-effect free methods when building our set S.

However, although a side-effect free method may not advance the state of an object, it may still have sequencing constraints; for example, a method to get a value may not be invokable before the value has been set. We thus define for each side-effect free method the set of states for which it is legal to call that method. We represent this in our model diagrams with dotted lines connecting methods that are not states to the states in which that method is legal.

To illustrate this, let us consider again the example from the previous section, and assume that `get_a` and `get_b` do not have side effects. The final models are shown in Figure 4.

2.3 Formal definition of the model

We now expand our naïve definition of a model to include these two refinements:

Definition 2.2 An *interface model* of a class is a collection of submodels, each of which is a triple, denoted $M = \langle U, S, T \rangle$, where U is the set of methods governed by the submodel plus the special methods START and END, $S \subseteq U$ is the set of methods that qualify as states, and $T \subseteq (S \times U)$ is the set of allowable method pairs. An instance is in state *op* if *op* is the last method from the set S that has been invoked on the instance, or in state START if no such method exists. A method $op' \in U$ can be invoked on an instance if and only if the instance is in state *op* and $(op, op') \in T$.

3. STATIC MODEL EXTRACTOR

This section describes a static analysis technique that can automatically detect the sequencing constraints in component interfaces designed to guard against misuse. The basic idea is that we analyze the uses and definitions of fields in the class, and identify pairs of method invocations that would cause an exception in the program. For example, if one method stores a `null` to a field, it cannot be followed by an invocation to a method that dereferences that field unconditionally. We first discuss why this information is often available in well-designed Java components, and then discuss our technique that extracts such information.

3.1 Defensive Programming

Widely used components are often defensively programmed to detect illegal sequences of method invocations so as to ensure the integrity of a component. A typical approach is to maintain a state value and then check if the state is valid before performing an operation. If it is not valid, a Java program will usually throw an appropriate exception; programs written in other languages would typically return a return code indicating an error or would assert that the state is invalid and exit the program.

The state may be implicitly encoded in a field that holds the normal data being operated on. For example, a null field would indicate that the method that sets it has not been invoked. Sometimes, the state is explicitly encoded in a field. This field is typically implemented as an enumerated type holding all the possible states the object is in. Without the benefit of enumerated types, Java implementations typically use integers or booleans. The field is similar to the concept of type state variables[22]. This paper presents empirical evidence that the notion of type state variables is particularly applicable to object-oriented software, and especially to languages that have a good exception handling facility.

Let us consider as an example the design of a list iterator, as shown in Figure 5. This code is taken from the `java.util.AbstractList.ListItr` class in the standard Java 1.3.1 class library. `java.util.AbstractList.ListItr` implements the `java.util.ListIterator` interface. The `set` operation writes some object to the "current" element of the iterator, as indicated by the index variable `lastRet`. The `next` or `previous` methods adjust the index variable, whereas the `remove` or `add` methods eliminate the notion of having a current element. The latter is indicated by setting the variable `lastRet` to -1, which will cause the `set` operation to throw an `IllegalStateException` if there is no intervening `next` or `previous` method invocations. The `lastRet` variable thus doubles as a value used in the algorithm as well as a type state.

For code that has been defensively programmed, it is possible to analyze the implementation to deduce sequences that are forbidden by the component.

3.2 Algorithm

Our algorithm consists of three steps. First, for each method m, we identify those fields and predicates that guard whether exceptions can be thrown. Second, we find those methods m' that set these fields to values that can cause the exception. Immediate transitions from m' to m are illegal. Finally, an interprocedural analysis is run over the class to determine which methods may modify or reference the field. The complement of the illegal transitions with respect to the relevant methods forms a model of transitions accepted by the static analysis.

```
public Object next() {
    // ...
    lastRet = cursor++;
    // ...
}
public Object previous() {
    // ...
    lastRet = cursor;
    // ...
}
public void remove() {
    if (lastRet == -1)
        throw new IllegalStateException();
    // ...
    lastRet = -1;
    // ...
}
public void add(Object o) {
    // ...
    lastRet = -1;
    // ...
}
public void set(Object o) {
    if (lastRet == -1)
        throw new IllegalStateException();
    // ...
}
```

Figure 5: Example code from `AbstractList.ListItr`.

The general problem of whether the invocation of a method may cause another to raise exceptions is undecidable. Fortunately, state variables are often used in a rather simple manner, as illustrated in the example above. Thus, we restrict our analysis to finding those simple predicates that involve testing a field with a constant null or a constant integer. Note that in Java, all dereferences are preceded with a null check, thus this definition includes finding the runtime exceptions thrown as a result of Java semantics.

First, the algorithm finds predicates that control whether or not exceptions are thrown. The algorithm computes the control dependence information[10] for each method. Then, for each site that can throw an exception, it checks if the predicate guarding its execution is a single comparison between a field of the current object and a constant value, and that the field is not written prior to being tested. If so, we say that that single predicate is a pre-condition for that method, and that the field being tested is a state variable.

The second step analyzes all the methods to find out if they *must* assign some constant values to the state variables. That is, we perform a constant propagation analysis on the methods and determine if the state variables are set to some constant at the exit. We have identified an illegal transition if the constant value satisfies the condition that guards the exception-throwing statements.

We have implemented these algorithms using the joeq compiler system[25]. It correctly identifies `start` → `set`, `start` → `remove`, `remove` → `set`, `add` → `set`, `remove` → `remove`, and `add` → `remove` as transitions that will throw an exception. The complement—that is, the allowable transitions—is shown in Fig. 6.

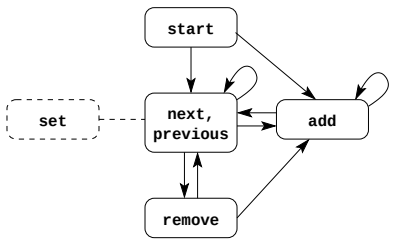

Figure 6: Submodel extracted statically from the slice on `lastRet` in `AbstractList.ListItr`.

4. THE DYNAMIC EXTRACTOR AND CHECKER

This section describes a dynamic instrumentation technique that can automatically detect the usage pattern of objects, building up a series of submodels, or can enforce an already specified model at run time. The basic idea is to track each instance of an instrumented class individually. When a method is invoked, we update the history of the object and record the sequence in each relevant submodel. We first discuss our basic extraction technique, then discuss some issues that must be addressed when implementing the system in Java.

4.1 Extracting a model

Given a set of class files to instrument, we begin by running the mod/ref analysis of Section 3 over the bytecode to determine which methods are relevant to each field. Each field f is assigned a submodel. Any method that writes f is deemed state-modifying, while any method that reads but does not write f is deemed relevant, but state-preserving. We then use this to guide the insertion of calls to our analysis routines at method boundaries.

Our system uses the Byte Code Engineering Library[6] to rewrite the bytecodes to insert the calls to our analysis routines. The analysis routines only deal with one submodel at a time; if a method is relevant to multiple models, multiple calls are inserted. Once all the calls have been placed, we then run training programs that use our instrumented objects.

Each instance keeps track of, for each submodel, that last state-modifying method that was called on that instance. When a method is invoked on an object, the extractor examines these records, updates the submodels appropriately, and then updates the last-call information of the instance for each submodel where the method is state-modifying. In order to capture the exported interface and ignore the internal usage of the object, the dynamic system maintains knowledge of the local call stack and ignores any call that is internal to that instance. Once the program finishes, transitions to the END state are noted for each instance, and the final set of models is written out.

The dynamic checker accepts a set of models as input. It is nearly identical to the extractor; the primary difference is that where the extractor adds elements to the set of legal transitions, the checker prints warnings or raises exceptions when a previously unseen transition occurs.

4.2 Complications

Though conceptually simple, a number of challenges and

Program	Description	Lines of Code	Analyses
`java.net` 1.3.1	Networking library	12,000	static & dynamic
Java libraries 1.3.1	General purpose library	300,000	static
J2EE 1.2.1	Business platform	900,000	dynamic
joeq	Java JVM implementation	65,000	static & dynamic

Table 1: Applications Used in the Experiments

complications arise when implementing this system.

Inheritance. A submodel specifies which methods to instrument in a class. However, some of these methods may not actually exist in the `.class` file because they are inherited from a superclass. We solve this problem by adding dummy methods that simply call the methods of the superclass explicitly. These may then be instrumented along with the methods that the class itself actually defines.

Exceptional control flow. Control flow may leave a method via an uncaught exception. Therefore, it is not sufficient to merely insert calls at the beginning of each method and before each return instruction. To deal with uncaught exceptions, we add an additional exception handler to each method that will catch any uncaught exception, call our analysis routines, and then rethrow the exception.

Multithreading. The Java language supports and encourages multithreading. This raises two issues; first, we must ensure that our model extraction routines do not get corrupted due to race conditions, and second, we must address the issue what a "last call" means when multiple threads are making concurrent calls on an object.

To ensure that model state is not corrupted, only one thread should actually be in the model extracting code at any given time. We ensure mutual exclusion by declaring the extraction/checking routines to be `synchronized`. Since these routines are fairly short and only occur at method boundaries, this does not significantly impact program execution.

When an object is being used by multiple threads, our system keeps a separate call history for each thread that accesses an object, thus capturing the protocol followed by each thread independently.

Memory leaks. We may not maintain any hard references to any object in the actual program, because this will preclude otherwise unreachable storage from being reclaimed by the garbage collector. We solve this problem by taking advantage of the Java 1.2 facility of *weak references* to refer to an object without inhibiting its reclamation by the garbage collector. When an object loses all non-weak references, the garbage collector notifies the dynamic system, which then adds an END transition and ceases tracking the object.

5. EXPERIENCES

We applied our tools to several real-life applications. The set of programs are shown in Table 1. We first use a model extracted from the `java.net` library v1.3.1 to illustrate our model. We show that the model provides significantly better results than the naïve one where all the methods are considered state transformers in the same model. Second, we use our static analysis technique to extract the models from the Java standard library v1.3.1. Third, we apply the dynamic instrumentation tool to extract models from the J2EE application, a large application framework for which we had no prior knowledge about the implementation. We have developed a simple "test suite" generator, and the models extracted gives us useful insight on the completeness of the test suite. Finally, we apply the static and dynamic tools to the joeq program, which is a Java compiler and a Java virtual machine written in Java, and show the value that the automatic model extraction can bring to a system designer.

5.1 Illustration of our Model and Tools

Our first experiment is to demonstrate the feasibility of our model in capturing a component interface. To do that we apply our technique to the abstract class `java.net.SocketImpl` in the `java.net` library. The `java.net.SocketImpl` is a common superclass of all classes that implement sockets in Java. It is used to create both client and server sockets.

The class consists of 16 methods, shown in Figure 7. We ran two experiments. In the first experiment, we used the dynamic model extractor to find a lower bound on the model. We instrumented the Eteria IRC client and exercised it by connecting to an IRC server. The second experiment applied the static analysis tool to find an upper bound of the model.

Method	Description
`create`	creates datagram socket
`connect`	connects the socket
`bind`	binds the socket
`listen`	sets the maximum queue length for incoming connections
`accept`	accepts a connection
`getInputStream`	returns input stream view
`getOutputStream`	returns output stream view
`available`	returns the number of bytes that can be read without blocking
`close`	closes the socket
`shutdownInput`	shuts down input stream
`shutdownOutput`	shuts down output stream
`getFileDescriptor`	returns file descriptor
`getInetAddress`	returns address for this socket
`getPort`	returns the remote port number
`getLocalPort`	returns the local port number
`reset`	closes and reinitializes the socket

Figure 7: Important methods in `java.net.SocketImpl`

5.1.1 Experiment: the dynamic model extractor

We ran the dynamic tools first assuming the naïve model where every method call changes the state of the finite-state machine. We obtained the model in Figure 8. The model has 11 states and 17 edges. Most of the implied usage rules

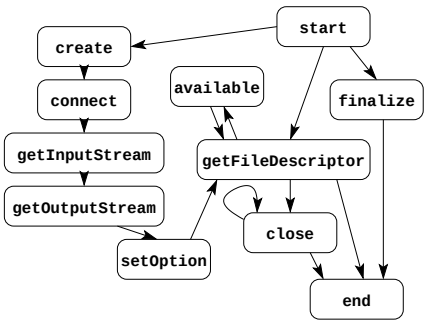

Figure 8: Dynamically extracted naïve model for `java.net.SocketImpl`

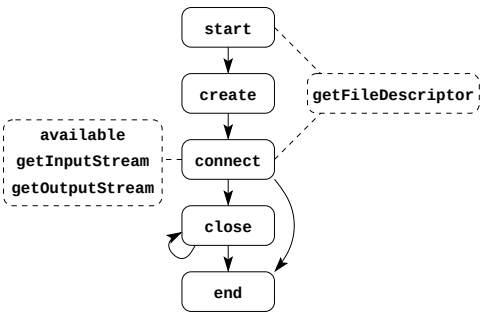

Figure 9: Dynamically extracted submodel of the `fd` field in `java.net.SocketImpl`

were simply artifacts in the client code — for example, we found that the only call that followed `getInputStream` was `getOutputStream`. This shows that a naïvely dynamically-extracted model, without slicing by fields or identifying state-preserving methods, is basically pointless in this case.

We proceeded to slice the methods by the fields that they access, separate out the state-preserving methods, and re-run the dynamic tests on these submodels. We obtained six models, five of which were simple two-state "put-get" models. The model for the `fd` file descriptor field was more interesting. The `fd` field has the following state-setting and state-dependent methods:

state-modifying: `create`, `connect`, `close`

state-dependent: `available`, `close`, `shutdownInput`, `getInputStream`, `getOutputStream`, `shutdownOutput`, `getFileDescriptor`

We re-ran the dynamic tests using this information, and obtained the model in Figure 9.

Slicing by fields and separating state-preserving methods produced a dramatically simpler model—one which succinctly summarizes the sequencing constraints on the object.

5.1.2 Experiment: the static analysis tool

In the second experiment, we applied our static analysis tool to determine transitions that would throw an exception. Our static analysis correctly identified that it was illegal to perform the sequences `(start)` → `getInputStream`, `(start)` → `getOutputStream`, `(start)` → `available`, `close` → `getInputStream`, `close` → `getOutputStream`, and `close` → `available`.

It is worth noting that the standard Java API documentation for the `java.net.SocketImpl` class does not specify under what conditions an exception would be thrown. It was only through inspection of the source code that we were able to check the results of our tool.

Inspecting the source code, we found that the `connect` method implicitly performs a `create` operation, therefore the `create` is unnecessary. Furthermore, it is legal to call `connect` multiple times, to call `create` after `connect`, and also to call `close` at any time. These transitions were not triggered in our dynamic tests, but our static analysis (and subsequent inspection of the source code) did confirm that these are in fact legal.

5.2 Automatic static model extraction: Java standard class library

To evaluate the effectiveness of our static analysis in extracting models from bytecode directly, we applied our tool to the Java standard class library v1.3.1. This library contains 914 classes, out of which the tool found 81 classes that had method sequences that invariably threw exceptions.

A wide variety of classes are amenable to this technique. The tool was able to identify seven iterator classes in the library as following the iterator model as discussed in Section 3. In addition, it also finds conformance to our model in many other classes:

`Vector` and `LinkedList`: Data cannot be retrieved if none exists. Calling a retrieval method immediately after construction or clearing the collection is illegal.

I/O and socket classes: Data cannot be read or written before the connection or the filestream are set up or after the file streams are closed.

`Timer`: New tasks cannot be scheduled if the timer has been canceled or finalized.

`SimpleTimeZone`: Certain methods cannot be accessed following some initializations.

`AlgorithmParameters`, `KeyStore`, `SecureClassLoader`, and `ClassLoader` classes: Attempts to use the object before initialization or reinitialize an already-initialized object throw an exception.

`ThreadGroup`: Attempting to destroy a `ThreadGroup` more than once throws an exception.

`Signature`: Initialization, updating and signing must proceed sequentially.

The actual constraints of the classes that we have extracted are shown in Figure 11. For brevity and clarity, we have consolidated and simplified the extracted constraints. Each row of the table describes the constraints that we found for different sets of classes; classes that were found to have similar constraints are merged into a single row in the table. The table contains two types of constraints. Negative constraints, marked with a "−" sign, say that calling a method in the "from" column immediately followed by calling a method in the "to" column will always throw an exception, and is therefore illegal. Positive constraints, marked

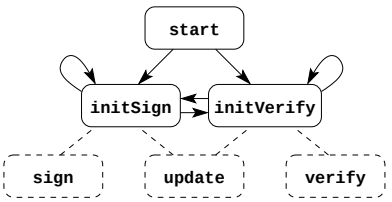

Figure 10: Model extracted statically from `Signature`.

with a "+" sign, say that the call in the "from" column *must* precede any of the calls in the "to" column, or an exception will be thrown. The last column in the table gives the condition on the instance variable by which the constraint was found.

The model extracted for `java.security.Signature` is particularly interesting. The state of the object is explicitly encoded by a field called `state` in the class. This `state` field is an integer whose value can be one of 0, 2 or 3.[1] The state is set to 0 in the initializer. Calling `initSign()` sets the state to 2, and calling `initVerify()` sets the state to 3. If `sign()` is called without being in state 2, an exception is thrown. Likewise, if `verify()` is called without being in state 3, an exception is thrown. If `update()` is called without being in state 2 or 3, an exception is thrown. Using this information, our static analysis was able to generate the model shown in Figure 10.

5.3 Evaluating Test Suites: J2EE

Our third experiment attempted to use the dynamic model extractor to characterize a test suite. In addition, we also wished to find the limit of dynamic instrumentation by exercising it on a large application. For this purpose, we selected the Java 2 Enterprise Edition (J2EE) version 1.2.1[18], a popular component architecture for creating multitier enterprise applications in Java. It is a large system comprising nearly a million lines of code in over 5,000 source files. As an application for the J2EE architecture, we used the Java Pet Store demo[19], a web enterprise application provided by Sun and intended to be used as a framework for developing J2EE business applications.

We have found that instrumenting every class of J2EE renders the system too slow to be usable. We identified the web service, the `org.apache` hierarchy, to be the bottleneck, and found that just skipping the instrumentation for that package is sufficient to create a usable system. The typical latency for loading a page was in the 5-10 second range.

The goal of this experiment is to determine if we can use the dynamic tool to evaluate a test suite. Unfortunately, we do not have access to an official test suite for the J2EE platform. For the purpose of our experiment, we have developed a tool that automatically generates test cases for web applications with a graphical user interface. This "button pusher" simulates a user traversing a web site by parsing and traversing the web pages. It randomly clicks on links, goes back, aborts transfers, reloads pages, etc. It also parses form data so that it can correctly fill out web forms. This tool was adapted from Ralf Wiebicke's *LinkVerify* program[26].

[1]Presumably, they used 0 rather than 1 to avoid having to initialize the state variable.

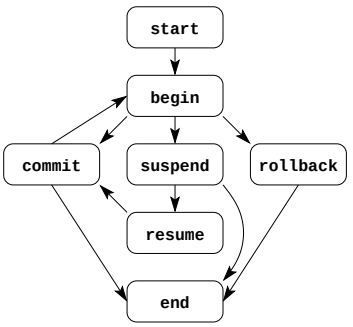

Figure 12: Sample model: TransactionManager

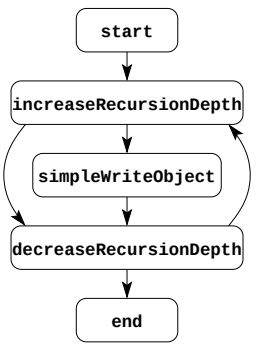

Figure 13: Sample model: IIOPOutputStream

Running multiple concurrent copies of our button pusher simulates a web server under heavy load. We collected our models by concurrently running two instances of the button-pusher and running each instance for two full iterations. A complete test run over the pet store took approximately 20 minutes.

To get a sense of how the button pusher compared to manual testing, we also performed a separate experiment by manually accessing the server from about a dozen different machines simultaneously and performing various operations. By comparing the models derived by these two testing methods, we found that the button pusher was much more complete and effective. In fact, the button pusher found several sequences of commands that triggered uncaught exceptions in the J2EE system.

We found that running the button pusher over the sample Pet Store application invoked just under a fifth of the methods in the J2EE library, and extracted models for 657 classes. Many of these, as we expected, have rather uninteresting models. For example, many of the models merely specified an initializer (often the constructor itself) followed by arbitrary sequences of different `get` and `set` routines. There were, however, approximately 50 very interesting models extracted, many of which had to do with database transactions.

As an example, the interface automatically extracted for the class `javax.transaction.TransactionManager` is shown in Figure 12. This example illustrates that the tool is effective in extracting sensible models automatically from Java byte codes. This uses our naïve model, because the class in question is an abstract interface, with no internal

class	+/-	from	to	variable
AbstractList.Itr AbstractList.ListItr Collections.5 HashMap.HashIterator HashMap.Enumerator LinkedList.ListItr TreeMap.Iterator	 + - 	 next, previous add, remove 	 remove, set remove, set 	 lastRet==null, index==-1
Vector LinkedList	-	init, removeAllElements	getFirst, getLast	size==0
i/o & sockets	+	close, finalize	any other methods	fd==null
PipedInputStream PipedReader	+ - -	connect connect receiveLast, close	receive, read connect receive, read	 connected==0,1
PipedOutputStream PipedWriter	+ +	connect connect	write connect	connected==0,1
ObjectInputStream	+	readObject, inputObject	defaultReadObject, readFields	currentObject==null
ObjectOutputStream	+ +	writeObject, outputObject putFields	defaultWriteObject, putFields writeFields	currentObject==null currentPutFields==null
Timer	-	cancel, finalize	sched	newTasksMayBeScheduled==0
SimpleTimeZone	-	some init function	decodeStartRule, decodeEndRule	various
AlgorithmParameters KeyStore SecureClassLoader ClassLoader	 + -	 init init	 "use" methods init	 initialized==0
ThreadGroup	-	destroy	destroy	destroyed==0
Signature	+ + +	start initSign, sign initVerify, verify	initSign, initVerify sign, update, initSign, initVerify verify, update, initSign, initVerify	 state==0,2,3

Figure 11: Constraints Extracted by Static Analysis

fields. A programmer unfamiliar with the system can learn some salient information about the code by just perusing these models. Developers may take this information as the starting point and augment it to create a complete model. More importantly, this also provides an interesting characterization of the behavior of the program or the test suite from which the model was generated. In this case, we see that rollback methods are always the last methods invoked on the object. This suggests that either the application simply discards all transactions once they were rolled back, or the test suite does not exercise any code that performs other operations after the rollback.

Shown in Figure 13 is another example that provides useful information about the test suite. The figure is the automatically extracted model for the J2EE internal class `com.sun.corba.ee.internal.io.IIOPOutputStream`. It is clear from the model that the recursion depth of the stream at any `simpleWriteObject` call is never anything other than 1. Thus, we see that our test suite does not adequately test the scenarios where recursion is encountered.

Our experiments with J2EE show that the automatic model extractor is capable of extracting useful object behavior from a large piece of software and also providing useful information that characterizes a test suite.

5.4 Protocol checking: joeq

Our fourth experiment is to explore if our tools can help a developer gain insight into his or her program and to find errors in the program. For this purpose, we applied our tool to the joeq system, which is a complete Java virtual machine and a Just-In-Time compiler developed by one of the authors of this paper[25]. Joeq can also compile Java `.class` files into Intel object code. The sources of joeq, consisting of approximately 65,000 lines of code, are available via `http://joeq.sourceforge.net`.

In joeq, there is a `jq_Method` object for every Java method in the system. As a method is loaded, prepared, and compiled in joeq, the corresponding `jq_Method` object goes through a sequence of states: `load` → `prepare` → `compile` → `execute`. Because of the complexity due to dynamic loading and linking, a large number of defects have been found in joeq that could be attributed to erroneous assumptions on the state that a `jq_Method` is in. In fact, as an aid to debugging, an explicit state variable has already been introduced to the `jq_Method` class so as to detect state violations at run time.

In this experiment, the developer of the software already has a reasonable model of the system. Our first step of the experiment is to use our dynamic model extractor to extract a model of the `jq_Method` class automatically. To our surprise, the model included several instances of a `load`→`compile` transition, which is a violation of the in-

tended API. Next, we ran our dynamic model checker on joeq, using as an input the intended API. This tool pinpointed the invocation of the `compile` method that created a violation. Further investigation revealed that a method in a subclass of `jq_Method`, `jq_InstanceMethod.setOffset`, actually *duplicated* the code in `jq_Method.prepare`. The assertions in the joeq source did not catch the lack of a `prepare` state because the `jq_InstanceMethod.setOffset` method also updates the state variable, indicating that `prepare` had been called. There was no external indication that `setOffset` actually performed the `prepare` operation. The program happens to work correctly, but it is very fragile: inadvertently calling `setOffset` may trigger an incorrect `prepare` transition. Furthermore, if the functionality of `prepare` is changed in the future, it is highly likely that `setOffset` would not be updated accordingly.

We also used the static analysis to analyze the behavior of the `jq_Method` class. It correctly identified the presence of an integer state variable, `state`, in the `jq_Method` class, and deduced the correct sequence, `load` → (`prepare` | `setOffset`) → `compile` → `execute`. The static analysis was also able to identify the "incorrect" `setOffset` transition. Submodels on other fields also identified data dependencies between methods; for example, the `execute` method dereferences the `compiled_code` field, which is only set in the `compile` method. The submodel for the `compiled_code` field correctly identifies this dependency.

This example illustrates the usefulness of an independent tool that checks the behavior of the program—it found a discrepancy between the intended and the implemented API, which may be a potential source of errors as the software evolves. In this case, had the programmer provided the specification, it would have been incorrect. This example also illustrates the usefulness of a dynamic model checker.

6. RELATED WORK

Our work on the dynamic model extractor and modeler was inspired in part by Ernst's Daikon system[9], which extracts invariants from programs dynamically. While Daikon tries to discover invariants such as relations between variables in a program, our system uses dynamic techniques to discover the component interfaces. Whereas Daikon has only been applied to relatively small programs, our instrumented code has much lower overhead and is applicable to large applications. Recently, Daikon and the static checker ESC have been integrated into a system such that Daikon can discover invariants that ESC tries to verify[20].

Other dynamic techniques include the DIDUCE system, which instruments data locations and tracks changes in the invariants as time goes by. This system has proven successful at finding the sources of errors in difficult corner cases[13].

Using finite-state machine models to model program behavior is quite common. The Metal system[5, 8] and the SLAM toolkit[2] have been very successful in applying FSM models statically to operating system code. Programming languages such as Vault[7], NIL, and Hermes[23] encode these machines directly into the source code. Systems such as PREfix also contain models that can be represented as FSMs[3].

The Metal system uses a simple global FSM model to track changes in state[5, 8]. It uses a metalevel compilation step to statically identify locations in the code where the model may be violated. Our models differ from the Metal system's in that we model object-oriented behavior rather than low-level system resources, and therefore we use per-instance FSM models rather than a single global FSM model.

The SLAM toolkit checks temporal safety properties of general C programs[2]. These properties are specified in a language called SLIC, which uses a safety automaton to model execution behavior at the level of function calls. The authors found that it works well for programs whose behavior is governed by an underlying finite-state protocol. Like us, they also found that splitting models based on data was effective in isolating behaviors.

Ammons et al. have a system for specification mining, which attempts to extract FSM models of program behavior from program traces[1]. Their system runs processed traces of C program code through an off-the-shelf probabilistic NFA learner. Our system, in contrast, trades generality in order to leverage object-oriented program designs to produce more focussed specifications of component behavior.

Our models of component behavior are very similar to the notion of typestates[22, 23]. In typestates, the type of the object changes as the values stored in its fields change or as the program invokes operations on the object. Strom and Yemini developed a system called Typestate and two languages, NIL and Hermes, that use formal typestate rules to enforce static invariants on the state of a program at different program points. They do not support traditional pointers. Programs are checked for conformity by using a flow-sensitive data-flow analysis[23] or a demand-driven backward analysis[22]. They used typestates to statically verify the initialization properties of values[23]. Xu uses typestates to check the safety of machine code[27].

Vault is a type-safe variant of C[7]. Vault requires that the programmer annotate or rewrite their code to match the strict type model. Vault has no notion of path-sensitivity, so the state of objects must be consistent at every point in the program. It also has many restrictions on aliasing. This allows Vault to make many strong assertions about program behavior, but many programs are written such that their roles are guarded by predicates. Vault's system forbids many valid programs written using this style.

Systems like Vault, NIL, and Hermes are complementary to our system. They are able to take advantage of strong models because they require the programmer to write their code to obey a strict model. Our technique is able to obtain weaker models over arbitrary, large pieces of code.

Our technique is similar to other notions of type. Girard introduces the notion of linear logic, which says that resources can be used only once[11]. Wadler uses linear logic to propose a new type system for functional languages that models changes of state[24]. Gifford and Lucassen introduced a type and effect discipline that uses type inference techniques to track accesses to resources[17].

Our ideas about models of objects map closely to the object-oriented design notion of roles. A role is the part of an object that that fulfills its responsibilities to other objects. There has been work on specifying role models[12] and mechanisms for role-based programming[14]. Kuncak et al. describe a system for the programmer to specify the roles of objects by their aliasing relationships with other objects, along with a mechanism to statically verify those aliasing constraints[15].

7. CONCLUSIONS

This paper proposes a new model for component interfaces. We propose using a finite state machine for each field of a class, with one state for each method that writes that field. We then add restrictions on from which states methods that read the field may be invoked. We have proposed and implemented two techniques that automatically extract such models. The first is a dynamic instrumentation technique that records legal method sequences from working programs. The second is a static analysis that infers pairs of methods that cannot be called consecutively. We have also developed a dynamic model enforcer to ensure that a given model is obeyed.

We have applied our tools to four large software systems and found that it can automatically extract a variety of useful data, assisting in various stages of software development. A programmer may run this tool to learn about the common uses of a component. One can build a model of a component by extracting the models implied by a test suite, or those resulting models may be examined to evaluate the completeness of the test suite itself. A component developer may also use these tools to determine if the implemented API matches the intended one.

8. REFERENCES

[1] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[2] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN 2001 Workshop on Model Checking of Software*, pages 103–122, 2001.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience (SPE)*, 30:775–802, 2000.

[4] A. N. Habermann R. H. Campbell. The specification of process synchronization by path expressions. *Lecture Notes on Computer Science*, 16, 1974.

[5] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 59–70, 2000.

[6] M. Dahm. Byte code engineering library. http://bcel.sourceforge.net, 2000.

[7] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 1–16, 2000.

[9] M. Ernst, J. Cockrell, W. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 213–224, 1999.

[10] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[11] J. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[12] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.

[13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, 2002.

[14] M. Van Hilst and D. Notkin. Using C++ templates to implement role-based designs. Technical Report TR 95-07-02, Department of Computer Science and Engineering, University of Washington, 1996.

[15] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, pages 17–32, 2002.

[16] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proceedings of the International Symposium on Computer Architecture*, pages 192–203, 2001.

[17] J. Lucassen and D. Gifford. Polymorphic effect systems. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[18] Sun Microsystems. Java 2 platform, enterprise edition. http://java.sun.com/j2ee/, 2001.

[19] Sun Microsystems. Petstore application for the java 2 platform, enterprise edition. http://java.sun.com/features/2001/05/petstore.html, 2001.

[20] J. Nimmer and M. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Electronic Notes in Theoretical Computer Science*, volume 55, 2001.

[21] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, 2001.

[22] R. Strom and D. Yellin. Extending typestate checking using conditional liveness analysis. *Software Engineering*, 19(5):478–485, 1993.

[23] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *Software Engineering*, 12(1):157–171, 1986.

[24] P. Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, 1990.

[25] J. Whaley. The joeq virtual machine. http://sourceforge.net/projects/joeq, 2001.

[26] R. Wiebicke. Linkverify. http://rw7.de/ralf/htmltools/index.en.html, 1998.

[27] Z. Xu, T. Reps, and B. Miller. Typestate checking of machine code. In *Proceedings of the European Symposium on Programming*, pages 335–351, 2001.