

Application of Automated Environment Generation to Commercial Software

Oksana Tkachuk
Fujitsu Laboratories of America
Sunnyvale, CA
oksana@fla.fujitsu.com

Sreeranga P. Rajan
Fujitsu Laboratories of America
Sunnyvale, CA
sree@fla.fujitsu.com

ABSTRACT

Model checking can be an effective technique for detecting concurrency-related errors in software systems. However, due to scalability issues, to handle industrial-strength software, model checking needs to be combined with powerful reduction techniques. In this work, we applied modular model checking to detect errors in Interstage Business Process Management (I-BPM) software, a Java client-server application spanning more than 500,000 lines of code. To model check a separate module, one needs to represent its context of execution, i.e., its *environment*. Environment generation is a significant challenge, since the environment is to be general enough to uncover the module's errors, yet restrictive enough to allow tractable model checking.

In this paper, we present an experimental study that demonstrates the effectiveness of environment generation as a reduction technique in general and the effectiveness of automated environment generation in particular. Since model checking of the original application was intractable, we compared performance of automatically generated environments to environments written previously by hand. Automatic environments were obtained using Bandera Environment Generator (BEG). We present results of modular verification for both manual and automatic environments. The results show that automatically generated environments produce systems with a smaller state space, yet, for faulty modules, uncover the errors and, for error-free modules, produce coverage similar to manual environments.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Model Checking*

General Terms

Verification

Keywords

Modular Model Checking, Environment Generation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA '06, July 17–20, 2006, Portland, Maine, USA.
Copyright 2006 ACM 1-59593-263-1/06/0007 ...\$5.00.

1. INTRODUCTION

Recent research efforts [9, 31, 33] show that model checking can be an effective technique for detecting concurrency-related errors in software systems. However, due to scalability issues, to handle industrial-strength software, model checking needs to be combined with powerful reduction techniques such as partial order reduction [26], data abstraction [10], predicate abstraction [3], slicing [16], heuristic search [15], and modular model checking. In this paper, we pursue and evaluate the modular approach, which restricts analysis to selected parts of a program.

To model check a module in isolation, one has to model its context of execution, i.e., its *environment*. Environment generation is a significant challenge, since the environment is to be general enough to cover interesting module behaviors and uncover errors, yet to be restrictive enough to allow tractable model checking, yet not so restrictive as to miss important behaviors and mask errors. Experience shows that environment generation is largely done either by hand (e.g., [22]) or not done at all. For example, in [21], while model checking Linux kernel's TCP protocol, due to complexities of modeling interactions between the protocol and the kernel, a decision was made to run the entire Linux kernel in a model checker. This is understandable, since interactions between a module and its environment can be complicated and difficult to analyze: the environment can influence the module's control (e.g., by invoking the module's methods) and data (e.g., by modifying the module's data flowing into the environment). In Java, other interactions can happen through synchronization, exceptions, and global references.

Environment generation is a problem persistent across different types of program analysis: in unit testing, one has to write test *drivers* or supply test suites; in static analysis, one has to supply analysis results for components that are missing or impossible to analyze (e.g., *stubs* for native methods in Java); in modular model checking, one has to write both drivers and stubs.

Several approaches exist to modeling drivers and stubs. In unit testing, there are tools (e.g., JTest [18]) that use simple structural analysis of a Java class under test to automatically generate JUnit [19] tests. However, automatically generated JUnit tests are limited to sequential drivers that perform short sequences of method calls, with simple parameter values. User assumptions can be used to specify more complicated sequences of actions a driver may perform on a module (e.g., LTL is used in [23] and regular expressions in [2]). More sophisticated static analyses can be used to identify environment data more precisely (e.g., in [27], a

data analysis is used to identify partitions over environment data). Section 7 presents more related work, however, in spite of many approaches that can be used for environment generation, there is a lack of comprehensive automated support and case studies that give insight into when and why a particular approach or a combination of approaches can be applied to real software.

In this paper, we present an experimental study that demonstrates the effectiveness of environment generation as a reduction technique in general and the effectiveness of automated environment generation in particular. As a case study, we used a development version of Fujitsu enterprise software called Interstage Business Process Management (I-BPM), a Java client-server application, which at the time contained real errors. When we were ready to test automated environment generation for I-BPM, several concurrency-related errors were already found using environments written previously by hand [17]. Manual environment generation was done by writing drivers for two modules and by manually modeling the parts of the application that prevented tractable model checking (e.g., RMI and JDBC). When we set out to perform automated environment generation for the case study, we wanted to find out (1) whether automated environment generation would work for this application; (2) what factors in module-environment interactions enable or prevent automated environment generation; and (3) how the automatic environments compare to the manually written ones, i.e., what measure can be used to evaluate environment's "goodness".

To generate environments, we used Bandera Environment Generator (BEG) [29, 30], which supports both *synthesis* of environment models from user assumptions and *extraction* of environment models from environment implementation using static analysis techniques. To model check a module combined with its environment, we used Java PathFinder (JPF) [6], an explicit state model checker built on top of a customized virtual machine. JPF can execute a Java program along all possible paths, checking for run-time errors (e.g., unhandled exceptions) and synchronization-related errors (e.g., deadlocks and race conditions).

This paper makes several contributions including (1) successful application of automated environment generation to a real industrial-size application; (2) evaluation of module-environment interactions that enabled the technique; and (3) comparison of automatically generated environments to the manual ones. To measure "goodness" of environments, we ran experiments for both faulty and error-free versions of the modules and measured branch coverage. The results show that automatically generated environments produce systems with a smaller state space, yet, for faulty modules, uncover the errors and, for error-free modules, produce coverage similar to manual environments. In addition, we show off benefits of automatic environments by comparing them to the manually written ones in terms of effort it took to write them. The paper gives insight into what worked, what did not work, why, and how the limitations of the current approach can be addressed in the future.

The next section presents the high-level architecture of the I-BPM application. Section 3 describes methodology we used for manual and automatic environment generation. Sections 4 and 5 present experiments for the two modules we analyzed: database adapter and cache. Discussion on the learned lessons and possible future directions is presented in

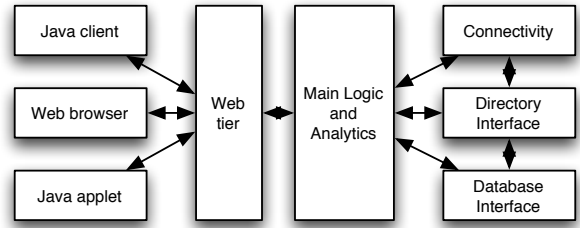


Figure 1: I-BPM Architecture

Section 6. Section 7 describes the related work and section 8 concludes.

2. I-BPM APPLICATION

The I-BPM application has a 4-tier architecture, as shown in Figure 1. The user interface can be implemented as a Java client, Java applet, or as a web browser. The main process logic and the analytics engine reside in the third tier. The fourth tier contains the underlying repositories including a database, directory, and document management. The I-BPM modules communicate through Java RMI.

The version of the application we examined contained over 1700 classes, spanning over 500,000 lines of code (LOC). We worked with the same version that was used for manual environment generation in [17]. The two modules that were analyzed were database adapter and cache, both residing in the third tier.

In the next section, we describe approaches used for manual and automatic environment generation.

3. METHODOLOGY

In this work, we consider decomposition of a Java program into a module under analysis and its environment. The module under analysis can be any collection of Java classes, preferably, a cohesive collection with well-defined APIs. The environment is comprised of the remaining classes with which the module interacts. The environment classes are broken into drivers and stubs. Drivers are Java classes that hold a thread of control (e.g., classes with the `main` method or threads). Remaining environment classes are called stubs. The modular model checking methodology we use in this work includes the following steps:

1. property specification and isolation of a module under analysis
2. environment generation: drivers and stubs
3. model checking the module combined with its environment and analysis of the results

Though theoretically straight forward, the approach is extremely difficult in practice. One of the first obstacles is selection of the appropriate module and specification of its properties. Just having complaints from customers about a system "hanging" or "acting funny" is not sufficient to pinpoint the faulty modules. One may attempt to model check the entire program, however, in our case, I-BPM software is an open reactive system with a GUI and requires QA testers (and end-users) to click on screen buttons to exercise the program's behavior. Therefore, not only a sheer size

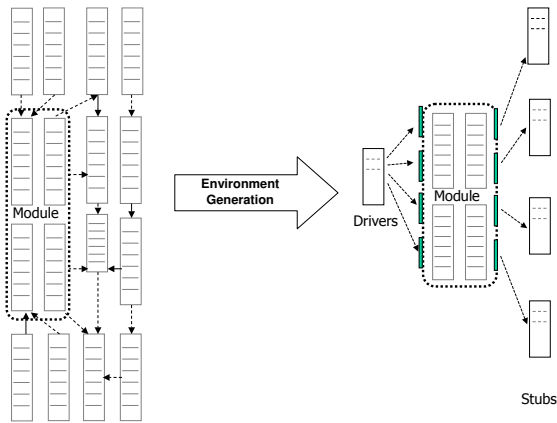


Figure 2: Environment Generation in BEG

of the application but its dependence on end-users requires environment generation.

To achieve step 1, we used input from the development team to pinpoint possible erroneous classes and inconsistencies in their behavior that allowed us to state their properties formally. The two modules were identified as database adapter, which included the `DbAdapterImpl` class, and cache module containing the `ProcessDefinitionProxy` class. At the time, it was not clear whether we would need to include more classes into the modules.

Once the module is identified, one has to write a driver to instantiate classes inside the module and appropriately exercise their behavior. In addition, since the code inside the module may reference other Java classes, their actual implementation or stubs need to be provided.

3.1 Manual Environment Generation

Since there was no documentation on how the modules were to be used, manual drivers were written using an iterative trial-and-error approach. For the database module, a driver was written to bring up the database server. It was tested and fixed after each time it failed (e.g., due to unset global values in the system).

Manual stubs were produced using the trial-and-error approach as well. Classes that prevented model checking or caused a state space explosion were replaced with simpler versions. The following changes were performed manually to enable model checking: (1) the RMI package was replaced with models that did not attempt to connect to a network; (2) JDBC and related database access classes were modeled using a hashtable; (3) the `Date` class was edited to always give the same time to avoid a counting variable in a state space; (4) localization classes were stubbed out.

It is hard to evaluate how much effort went into developing manual drivers and stubs. Writing manual environments took about a month by one person. This included a time it took to learn enough about the application to develop sensible environment code.

3.2 Automated Environment Generation

In this section, we describe capabilities of Bandera Environment Generator (BEG), used to generate environments for the database adapter and cache modules. Figure 2 depicts the environment generation approach implemented in

BEG. Given a collection of Java classes that make up the module, BEG automatically discovers the interface between the module and the environment and generates the module's drivers and stubs. The picture depicts a simplified case when drivers make calls to the module (showed by arrows pointed from the drivers to the module) and stubs are called by the module. In general, the interactions may be arbitrary (e.g., the stubs may have callbacks).

BEG supports generation of multi-threaded drivers from user specifications. Such drivers instantiate classes inside the module and perform sequences of method calls on those instances. BEG allows users to specify compactly the number of driver threads, the number of instantiations, and the sequences of method calls each thread performs on the module using LTL or regular expressions; method invocation and assignment expressions are used as atomic propositions. When generating Java code, each atomic proposition is translated into a legal Java assignment or a method invocation, including receiver objects and argument values. BEG supports specification of concrete and abstract values. In the next section, we show examples of specifications and drivers generated from them.

When writing stubs, BEG offers the following support. Given a set of classes that make up a module, BEG automatically identifies classes that are referenced inside the module and generates stubs for them based on results of static analysis of the environment implementation. BEG offers flexibility regarding how much behavior is to be included in stubs. One may start with simple, empty, stubs. If such stubs are not sufficient to find errors, one may turn on static analyses such as detection of callbacks from environment to the module and side-effects analysis, which calculates how the environment may modify the module's data. Side-effects analysis in BEG can be configured to detect useful properties like *containment*. In [11], such analysis was used to detect containment properties of `javax.swing` classes by calculating effects on their specific fields (e.g., arrays and lists), which can be used to store and retrieve data.

By design, to insure scalability and effective reductions, static analyses in BEG do not calculate all possible dependencies between a module and its environment. Therefore, stub generation in BEG is *not safe*, meaning that BEG may miss some stub behavior that may, in turn, influence the module behavior. For example, BEG does not include analyses that calculate if a stub may go into an infinite loop; stubs are assumed to always return. As such, stub generation is an effective technique for uncovering errors but may be unsafe for proving correctness (i.e., absence of errors). To tackle this problem, more analyses can be added to detect more module-environment interactions, e.g., divergence and concurrency-related dependencies. To calculate all possible dependencies, one can use *slicing* techniques, however, preliminary results show that slicing of concurrent Java programs is not as scalable and may preserve too many dependencies in the final slice.

3.3 Manual vs. Automatic Approach

When writing a driver, both manual and automatic approaches have a similar learning curve: one has to understand how a module is used to come up with the driver implementation or its corresponding high-level specification. One may question the efficiency of writing a specification compared to writing code directly. While writing specifi-

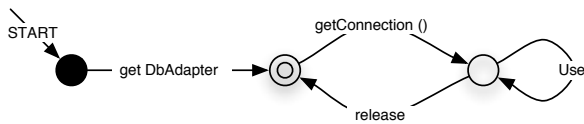


Figure 3: Database Adapter Protocol

cations gives small advantage in terms of time it takes to write them, it gives great advantage in terms of producing compact drivers. In all instances, we observed that manual drivers contained redundant computation and created unnecessary objects, while exercising the behavior of the modules along the same paths. In the next section, we present data that show the impact of automated drivers on the verification effort.

Comparing manual and automatic stub generation is easier. The manual strategy is to start with an original implementation of the environment and then abstract away details that prevent model checking (e.g., RMI and JDBC). The automatic stub generation is more aggressive. It identifies classes that are referenced inside the module and generates compact stubs for them, while accounting for the behavior of the rest of the environment. Therefore, the manual approach starts with the upper bound on the environment behavior and abstracts away the details that prevent model checking; the automated approach starts with the lower bound on the environment behavior and adds more details until an error is found or desired coverage is achieved.

In the next section, we compare manual environments to automatic ones in terms of their size, the effort it took to develop them, and the impact they had on verification of the two modules under analysis. All experiments were run on a desktop with 3G RAM, 1.6 MHz processor running Linux 9.1, using SDK 1.5.

4. DATABASE ADAPTER MODULE

The database adapter module is used to communicate with an underlying database, using an RMI protocol. The relevant parts of the protocol are shown in Figure 3. The client code:

1. acquires the `DbAdapter`
2. calls `getConnection()` on the `DbAdapter`, receiving a `DbConnection`
3. uses the database through the `DbConnection`
4. releases the connection

Creation of `DbConnection` objects is very slow. To save time, step 4 returns the `DbConnection` object back to a pool of available connections. If a client dies for some reason, while holding a connection, the connection is not returned to the pool. To fix this problem, I-BPM checks each of its connections in step 2 to see if the original client is still alive. If the original client seems dead, the connection is handed out to a new client. If a client is dead due to a broken network connection, this behavior is correct. However, in the implementation, the client is assumed dead if its connection is 5 minutes old:

```
long stale = System.currentTimeMillis() - 300000;
if (stale > conn.getLastUse())
    releaseConnection(conn);
```

```
environment{
  instantiations{
    DbAdapterImpl adapter = new DbAdapterImpl("", 3);
    connect("", "", "", "");
  }
  driver-assumptions{
    2 Client: getConnection (); releaseConnection() #
  }
}
```

Figure 4: User Assumptions

```
public class EnvDriver {
  public static void main(java.lang.String[] param0){
    DbAdapterImpl adapter = new DbAdapterImpl("", 3);
    try{
      adapter.connect ("", "", "", "");

    }catch(Exception e){e.printStackTrace();}
    Client c1 = new Client(adapter).start();
    Client c2 = new Client(adapter).start();
  }
}

public class User extends java.lang.Thread {
  public DbAdapterImpl adapter;
  public Client(DbAdapterImpl param0){
    adapter = param0;
  }
  public void run(){
    try{
      adapter.getConnection();

      adapter.releaseConnection(
        (DbConnection)Verify.randomObject("DbConnection");
    }
    catch(Exception e ){e.printStackTrace();}
  }
}
```

Figure 5: Driver Models for Adapter Module

Therefore, it is possible for 2 clients to end up with the same `DbConnection` object. This may lead to unwanted interleaving of clients' transactions.

During manual environment generation, the above code in `DbAdapter` was modified to model the possibility of releasing connections using the modeling primitive `Verify.randomBool()`:

```
if(Verify.randomBool())
  releaseConnection(conn);
```

JPF interprets `Verify.randomBool()` as a nondeterministic choice between `true` and `false`, exploring both branches of the `if` statement. Identifying conditions that are influenced by the module's environment and need to be modeled using `Verify.randomBool()` is difficult, if done manually. Automated stub generation presented in section 4.3 solves this problem by inserting nondeterminism into stubs.

4.1 Properties

We restrict our presentation to checking properties that were found violated using manual environments. One such property was "no two clients get the same `DbConnection` object".

When generating environments with BEG, we started with a module consisting of the `DbAdapterImpl` class, which has 330 LOC and 12 public methods in its interface. We did not know if other classes (e.g., `DbConnectionImpl`) would need to get added to the module later.

4.2 Drivers

Using the protocol in Figure 3 as a guide, we started with a specification shown in Figure 4. The specification file contains two sections: an instantiation section, which shows

what objects to create, and an assumptions section, which shows the number of threads and how they exercise the behavior of the instantiated objects.

The methods used in the specification are the `DbAdapter` methods that implement the protocol in Figure 3. The constructor takes an integer parameter denoting the number of connections to be created; `connect()` creates the connections; `getConnection()` hands out an available connection; and `releaseConnection()` returns a connection back to the pool of available connections. The `DbAdapterImpl` class declares several fields used to store available and used connections. The protocol methods adjust these container fields accordingly.

When specifying a method call, one can omit values for receiver objects and parameter values. BEG fills the holes with values of appropriate types. For example, BEG uses the only instance of `DbAdapter` in the system as a receiver object for all calls to the module. If more than one instance exists or no information is available, BEG can be configured to generate `Verify.randomObject(String type)` expressions. Such expressions are interpreted by JPF as a nondeterministic choice over a set of available heap objects of the appropriate type. Figure 5 shows the Java code generated from the specification. Note that this particular driver, due to a nondeterministic choice over connection objects, includes executions where one client can release another client's connection. To avoid such behavior, we refined the driver by specifying concrete values for the connections:

```
2 Client: connection = getConnection();
   releaseConnection(connection) #
```

The property

```
assert
!(c1.connection != null && c2.connection != null)
|| (c1.connection != c2.connection);
```

is embedded at the end of the `main` method. It states that if two clients hold a connection (which is not `null`), then the connections should be different. We tested the driver combined with the faulty module and previously written manual stubs. To our surprise, the property did not get violated.

We increased the number of threads in the specification to 3, regenerated the driver, and the property got violated. The counterexample analysis revealed that the property violation occurs only when, upon calling `getConnection()`, there is only one available connection left. Therefore, such behavior manifests only if the number of clients is equal to or more than the number of created connections.

At this point we were ready to compare the automatic driver with the previously written manual one. To make comparison of drivers fair, we adjusted both manual and automatic drivers to create the same number of connections and to start up the same number of clients. Preliminary results showed that the manual environment was producing a system with huge state space.

We took a closer look at the manual driver: it set one of the debugging flags on. Even though the driver did not print any messages while running, it did call many debugging routines, checking whether some messages needed to be printed. We turned the debugging in the manual driver off and its performance increased by a factor of 5. We decided to compare automatic environments to the manual environments that did not make any calls to debugging or logging. We wanted to see what reductions, if any, the automatic environments could perform beyond abstraction of debugging.

The next round of comparison uncovered missing behavior in the automatic drivers: the branch coverage was lower than for the manual drivers. It turned out that the manual driver creates an instance of `DbAdapter` through a call to `DbAdapterFactory.createDbAdapter()`. The method creates an instance of the adapter, calls `connect()` followed by `disconnect()` on the adapter to test its functionality, and stores it into an RMI registry. Then it looks up the adapter through a call to `Naming.lookup()`:

```
public static void main(String[] args)
{
    DbAdapterFactory.createDbAdapter();
    //stores using
    //Naming.rebind(remoteName, dbAdapter);
    ...
    //retrieves
    DbAdapter adapter = (DbAdapter) Naming.lookup(
        remoteName);
}
```

Note that model checking programs with references to the actual `java.rmi.Naming` class is intractable. To enable model checking, a simple stub for the `Naming` class was written during manual environment generation. The `Naming` stub was modeled as a simple container with methods `rebind()` and `lookup()` used to store and retrieve adapter objects. We believe that manual stubbing out of the `Naming` class was performed to enable model checking rather than to preserve semantics of RMI. Since storing the adapter object in an array and retrieving it does not have any effects on the adapter objects, we decided not to refine BEG specifications with calls to the `DbAdapterFactory` and `Naming` classes, even though the specification language of BEG allowed it. In the end, we refined the specification by adding the missing calls to `connect()` followed by `disconnect()` to test the newly created instance of the adapter. The resulting automated driver performed the same sequences of calls to the `DbAdapter` as the manual one, while omitting calls to other classes in the environment.

4.3 Stubs

Stub generation is a challenging exercise. Different classes require different stub generation approaches, which depend on the types of properties one wants to preserve in the stubs. For example, stubs for `java.util` (e.g., `Hashtable` and `Vector`), used in the adapter module to store available and used connections, need to preserve containment properties, if one wants to keep track of how many available connections are left at each client request.

BEG can be configured to track containment properties and to generate containers that store data in an array. However, JPF handles original implementation of `Hashtable` and `Vector` without much overhead, and we decided to use container classes from `java.util` in their original form. In addition, many classes from `java.lang` (e.g., `java.lang.Object` and `java.lang.Thread`) did not need any modeling, because JPF has a built-in treatment for such classes.

To generate stubs, one has to feed module classes to BEG. In general, classes mentioned in a property should be included in the module. Since the database adapter property checks equality of connection objects, at first, we thought, we should include the `DbConnection` object into the module. However, since the connection objects are compared using object equality, rather than structurally, we thought we could stub out the `DbConnectionImpl` class and still verify the property.

Feeding the `DbAdapterImpl` class to BEG, we generated empty stubs using the simplest settings in BEG. Empty stubs return abstract values called TOP. Abstract values of primitive types are declared using a special class `Abstraction`, e.g., a stub for the `currentTimeMillis()` method, used by the `DbAdapter` to figure out connections' age, looks like

```
public class System{
  public static long currentTimeMillis(){
    return Abstraction.TOP_LONG;
  }
}
```

Reference types are treated in a special way. Each environment class declares a public static field called `TOP_OBJ`, e.g., treating `DbConnectionImpl` as part of the environment, BEG produces the following code

```
public class DbConnectionImpl implements DbConnection{
  public static DbConnectionImpl TOP_OBJ =
    new impl.DbConnectionImpl();
}
```

`TOP_OBJ` fields are used in simple stubs, for example, the `createDbConnection()` method, called inside the `connect()` method of `DbAdapter` to create a new connection, is stubbed out as follows

```
public class TransportFactory {
  public static synchronized DbConnection
  createDbConnection(java.sql.Connection param0){
    return DbConnectionImpl.TOP_OBJ;
  }
}
```

TOP values can be safely interpreted by abstraction engine or executed symbolically given support for symbolic execution. For example, given support for abstraction, a model checker can automatically infer that

```
long stale = System.currentTimeMillis() - 300000;
```

should be interpreted as `TOP_LONG`, and

```
stale > conn.getLastUse()
```

as `TOP_BOOL`, or `Verify.randomBool()`, thus, eliminating the need to identify conditions that may be influenced by environment.

JPF can be used to perform symbolic execution [20], however, we have not applied the technique in this work. Without support for abstraction or symbolic execution, TOP values are interpreted as regular values and one needs to check that they do not mask errors. The easiest way is to configure BEG to initialize `TOP_OBJ` fields to `null`. Testing with null values is useful to find out which objects flowing from environment are dereferenced or used otherwise in a setting that prohibits `null` values, e.g., creating `null` connections and inserting them into a hashtable of available connections raises an exception. Creating TOP connections is a problem as well, since TOP objects can not be compared deterministically. We already had doubts about how much behavior regarding `DbConnection` objects could be stubbed out. In the end, we added the `DbConnectionImpl` class to the module and reran the stub generation preserving side-effects with respect to `DbConnection` objects.

The side-effects analysis in BEG detects allocation sites for module objects and generates code accordingly. The refined implementation of the `createDbConnection()`, taking into account its effects on `DbConnection` objects, is

```
public static synchronized DbConnection
createDbConnection(java.sql.Connection param0){
  DbConnectionImpl dbconnectionimpl0 =
    new DbConnectionImpl(Connection.TOP_OBJ);
  return dbconnectionimpl0;
}
```

The refined stubs were sufficient to detect the error in the faulty module and, for error-free module, to produce coverage similar to manual environments.

Using `DbAdapterImpl` and `DbConnectionImpl` classes as input, BEG generates 125 classes with 2719 LOC. Stub generation for the adapter module runs within 2 minutes. During the manual effort, 11 classes with 1034 LOC were manually written. In addition, 15 classes with 5973 LOC were manually edited to enable model checking. We did not investigate the differences between the edited classes and their original implementation to count how many of 5973 LOC were written manually. It is also unclear how much time manual modeling took. The whole process took approximately one month by one person, however, it included the learning curve required to get familiar with the application.

4.4 Verification

Table 1 shows verification results for the adapter module with 2 connections and 2 clients. We ran the experiments on two versions of the example, one with errors, *Adapter_e*, and one with the errors fixed, *Adapter*. The latter is used to show impact of automatically generated environments on a full state space exploration; alternatively, we could force the model checker to perform a full state space exploration with a faulty version.

We ran JPF v3.1.2. To show influence of Partial Order Reduction (POR) on benefits of environment generation, we ran JPF with POR on and off. The rows in the table present different configurations of environments and JPF used: the *Mds* row presents model checking results for the module combined with the *Manual drivers and stubs*; the *AdMs* row shows results for *Automated drivers* combined with the module and *Manual stubs*; *Ads* stands for the use of *Automated drivers and stubs*; *P* indicates the use of *POR*.

The third and fourth columns in the table show numbers for states and transitions; such numbers remain constant across multiple JPF runs. Other numbers, like memory (in kb) and time (in hours:minutes:seconds format), may vary from run to run depending on JPF environment; we ran the experiments three times and calculated the average values. These numbers are representative but may be approximate. The column CE shows a counterexample length for the faulty module.

The data in the table show that automatic environments have a nontrivial reduction factor. Calculating the ratios Mds/Ads and $MdsP/AdsP$, using numbers of states, shows that automatic environments reduce the number of states for the adapter example by a factor ranging from 1.9 to 3.2. The third column in the table shows the values of Mds/Ads , entered to the left of the corresponding *Ads* entry. For example, the reduction factor of automated environment generation for the fault-free adapter module, with the use of POR, is calculated as $6,603/2,882$ and entered to the left of 2,882.

By design, automated drivers perform the same sequences of method calls to the module as the manual environments, thus, the path coverage for both environments is similar. In addition, we measured *branch coverage*, presented in the last two columns in table 1. Column Cov_M shows coverage for branches inside the `DbAdapterImpl` and `DbConnectionImpl` classes; column Cov_T shows total coverage for the entire example. The total coverage shows that the module closed with the manual environment has 600 branches, the mod-

Example	Config	Mds/Ads	States	Trans	Mem	Time	CE	Cov_M	Cov_T
<i>Adapter_e</i>	Mds		29,390	57,166	106,624	01:26	526	45/114, 4/156	132/600
	AdMs		23,762	42,594	104,448	00:44	521	45/114, 4/156	72/482
	Ads	1.9	15,384	32,976	41,344	00:13	516	45/114, 4/156	56/332
	MdsP		8,423	13,708	89,088	00:56	110		
	AdMsP		3,805	6,818	47,232	00:18	110		
	AdsP	3.2	2,599	4,762	41,088	00:10	99		
<i>Adapter</i>	Mds		26,709	53,749	104,896	01:15		38/114, 4/156	120/600
	AdMs		19,886	36,177	101,312	00:39		38/114, 4/156	55/482
	Ads	2.2	12,175	26,320	30,272	00:13		38/114, 4/156	48/332
	MdsP		6,603	11,835	90,048	00:44			
	AdMsP		4,055	8,041	56,000	00:22			
	AdsP	2.3	2,882	5,903	39,296	00:10			

Table 1: Verification Results for the Database Adapter Module

ule closed with automated drivers and manual stubs has 482 branches, and the module closed with the automated drivers and stubs has only 332 branches, out of which 114 branches belong to the `DbAdapterImpl` class and 156 branches belong to the `DbConnectionImpl` class. It is clear that automated environments add a small amount of branches to the system, while manual environments contain a larger number of decision points (in this case, $600 - 114 - 156 = 330$).

Examining branch coverage, we see that the automatic environments cover as many decision points inside the module as the manual ones. In addition, the data show that finding a counterexample for the faulty adapter module is more expensive than performing the whole state space exploration of the fault-free example. Examining branch coverage for the faulty module suggests that erroneous paths invoke extra code, which is not exercised in the error-free example.

5. CACHE MODULE

The purpose of the object cache is to ensure that database accesses occur as seldom as possible. We identified the `ProcessDefinitionProxy` class, called `PDProxy` in the rest of the paper, as the entry point into the module. The class acts as a layer between the main code and the database adapter; it has 1257 LOC and 54 public methods.

5.1 Properties

We restrict our presentation to the two properties that were found violated using manual environments. One of the properties checks for *race* conditions and the other checks the *consistency* between the cache and the database.

5.2 Drivers

Many methods of the `DbProxy` have redundant functionality. Therefore, only several methods that have different purposes (such as `edit()`, `commit()`, `cancel()`) were used to specify the drivers.

Specifying the kinds of drivers that were written manually was straight forward. To check for race condition, the manual driver could be specified using the following BEG specification

```
environment{
  instantiations{
    PDProxy p = makeNew();
  }
  driver-assumptions{
    Writer: edit(); comit(); destroy() #
  }
}
```

This driver creates an instance of the `PDProxy` class and spawns the `Writer` thread, which performs a simple sequence of calls to the only instance of the module available. The property is embedded at the end of the `main` method of the main thread and checks the consistency between two `Hashtable` fields of the `PDProxy` instance.

To check consistency between the cache and database, a manual driver created two instances of `PDProxy` and spawned two identical threads calling several methods in the module interface. The manual driver could be specified using the following BEG assumptions:

```
environment{
  instantiations{
    PDProxy p1 = makeNew();
    PDProxy p2 = makeNew();
  }
  driver-assumptions{
    CacheStresser1: p1.edit();
    p1.modifyProcessDefinition();
    (p1.commit() | p1.cancel()) #

    CacheStresser2: p2.edit();
    p2.modifyProcessDefinition();
    (p2.commit() | p2.cancel()) #
  }
}
```

The interesting part of driver generation for cache is generation of arguments for various method calls. The `makeNew()` method takes 3 parameters

```
public synchronized PDProxy
makeNew(PDStruct newStruct,
        UserAgentProxy userAgent,
        String clientContext) {}
```

To fill parameter values, BEG generates `TOP_OBJ` values of appropriate types. The `UserAgentProxy` class is an interface and can not be instantiated. To create an instance of the `UserAgentProxy`, BEG generates a simple stub for the `UserAgentProxyImpl` class.

The second property checks consistency between the contents of the field `pdStructShare` of the `PDProxy` and the database. Specifying this property required getting a handle on the database contents. The interface to the database is represented by the `UserAgentProxy` class, which is stubbed out during the driver generation part. We discuss the refinement of the `UserAgentProxy` class in the next section.

5.3 Stubs

First, using the `PDProxy` class as a module, we generated simple, empty, stubs. Such stubs were sufficient to find race conditions in a faulty cache module and produce good cover-

age for the race-free module. This step produced 93 classes with 1606 LOC.

Checking the second property required stubs refinement. First, since the field `pdStructShare` was of type `PDStruct`, we needed to include this class into the module and calculate potential environment side-effects on the instances of this class. Second, we needed to refine the model of the `UserAgentProxyImpl` class or include the original class into the module. The class implements methods that store objects of `PDStruct` type into the database, fetch them from the database, edit, commit, etc. Since BEG can be configured to detect containment properties, we ran BEG on the `UserAgentProxyImpl` class, calculating its effects on the `PDStruct` objects.

BEG could detect that storing an object into the database stores it into the field `procDefStruct` of the `PDTxn` object called `txn`. However, BEG could not detect that fetching the same object from the database would return an object that is reachable from `txn`. Instead BEG generates a new `PDStruct` object. Examining the code, we found that before retrieving an object from the database, it is cloned using

```
return (txn.cloneTransaction()).procDefStruct;
```

where the `cloneTransaction()` method creates a deep copy of `txn`, including a deep copy of its field `procDefStruct`. Due to cloning of objects, BEG is not able to pick up containment properties. One could use the results of BEG to model faithful stubs, including cloning of objects. However, we wanted to model the database as a simple container data structure. In the end, we wrote a simple stub for a database, using an array field to store objects.

Using `PDProxy` and `PDStruct` as module classes, BEG generates 92 classes with 1631 LOC, including one class with simple side-effects on `PDStruct` objects. All BEG runs in this section run within 3 minutes.

5.4 Verification

Table 2 shows verification results for checking race conditions (examples *Race_e* and *Race*) and consistency between the database and cache (*Consist_e* and *Consist*). Calculating the ratios Mds/Ads and $MdsP/AdsP$ for the *Race_e* and *Race* examples, shows reduction factors from 2.2 to 2.8, with reduction factors being slightly smaller when POR is on. For the *Consist_e* and *Consist* examples, the reduction factors range from 4.2 to 16.5, with reduction factors substantially smaller when POR is on.

The biggest reduction factors are seen in the last experiment, when checking the consistency property of the error-free cache module. We were surprised to see that most of the reduction in that example was caused by automated drivers (*Mds/AdMs*). Inspecting the differences, we found that manual drivers initialized the `PDProxy` objects in separate threads, whereas the BEG specification language suggested to the user that the set up could be done in the main thread, thus reducing thread interleavings during the initialization of the `UserAgentProxy`, `PDStruct`, and `PDProxy` objects.

The coverage numbers for automatic environments are consistent. All environments designed to check the race condition cover 28 branches inside the `PDProxy` class; environments designed to check consistency between the database and cache cover 44 (47 for the fault-free example) branches of the `PDProxy` class and 31 (36 for the fault-free example) of the `PDStruct` class. The branch coverage numbers are extremely small, however, both `PDProxy` and `PDStruct`

classes contain many redundant methods not used in the drivers. Also, we used coverage reported by the manual environments as the target coverage and stopped environment refinement when the target coverage was achieved.

6. DISCUSSION

During the course of the experiment, we asked several questions. Some of them got answered; others require more work. Here are the questions we answered:

Does BEG have sufficient support for writing drivers? We found BEG specification language capable of describing the kinds of drivers that were written previously by hand. However, it would be useful to extend BEG specification language to accept arbitrary fragments of Java code, including assertions.

Does BEG have sufficient support for discovering stubs? BEG offers flexibility in how much behavior is to be included in stubs. One may start with simple, empty, stubs. If such stubs are not sufficient to find errors, one may turn on one analysis, e.g., side-effects, and generate stubs with their data effects preserved. We were surprised that empty stubs enhanced with simple data effects were capable of uncovering errors. We were also surprised that stub generation could be done without much prior knowledge of the application.

How do automatic environments compare to manual ones? For all experiments, the automatic environments produced a smaller system, yet, for faulty modules, uncovered the errors and, for error-free modules, produced coverage numbers similar to manual environments. There are two components in BEG environments: drivers and stubs. BEG drivers perform better than manual drivers because they make calls to the module classes directly (e.g., in the adapter example). They also suggest to perform initialization of objects in the main thread (e.g., in the cache example), which avoids thread interleavings during initialization. BEG stubs perform better because they achieve higher level of abstraction.

What is a good measure for environment's quality? Environments generated by BEG may not be safe, i.e., in case of the "verified" result, we can not be sure that the program is free of errors. Therefore, we take a pragmatic approach to measuring quality of environments. We say the environment is "good" if it leads to a discovery of an error or, in case of the "verified" result, it shows good coverage for a module under analysis.

What was modeled manually and how these features were handled automatically? The following 4 features were modeled specifically during manual environment generation to enable model checking by JPF:

(1) RMI: The `java.lang.Naming` class, used to store and retrieve objects from the RMI registry, was modeled as a simple container. In the automatic drivers, the use of `Naming` class was omitted, as storing and retrieving of adapter objects from a container has no side-effects on them and, thus, does not influence the adapter properties checked in this paper.

(2) JDBC: For the adapter module, classes from `java.sql` used to implement JDBC were modeled as empty stubs. For the cache module, database behavior was stubbed out at the `UserAgentProxy` interface, using a simple container. BEG easily handled the first case, however, producing a simple container for the `UserAgentProxy` class proved to be difficult due to cloning of objects.

(3) Time: The `Date` class was manually stubbed out to

Example	Config	Mds/Ads	States	Trans	Mem	Time	CE	Cov_M	Cov_T
<i>Race_e</i>	Mds		1,800	2,935	42,688	00:07	653	28/468	128/1766
	AdMs		1,613	2,529	38,016	00:05	650	28/468	124/1714
	Ads	2.8	636	936	24,640	00:04	317	28/468	34/538
	MdsP		183	322	37184	00:06	36		
	AdMsP		109	185	27008	00:04	25		
	AdsP	2.6	71	114	19456	00:02	24		
<i>Race</i>	Mds		2,346	3,970	37,184	00:09		28/468	130/1766
	AdMs		2,148	3,425	34,560	00:07		28/468	126/1714
	Ads	2.2	1,057	1,690	26,304	00:07		28/468	36/538
	MdsP		500	948	24,856	00:06			
	AdMsP		287	525	21,944	00:04			
	AdsP	2.2	231	420	20,096	00:03			
<i>Consist_e</i>	Mds		123,587	210,097	109,340	01:15	1,318	44/468, 31/830	130/1750
	AdMs		20,662	38,905	65,344	00:22	1,310	44/468, 31/830	110/1714
	Ads	6.3	19,617	36,976	50,944	00:18	1197	44/468, 31/830	100/1374
	MdsP		31,285	53,185	65,264	00:29	195		
	AdMsP		8,018	14,946	58,304	00:18	478		
	AdsP	4.2	7,449	13,881	45,184	00:17	441		
<i>Consist</i>	Mds		24,811,745	69,958,785	705,472	05:54:14		47/468, 36/830	120/1750
	AdMs		1,568,157	4,286,581	219,456	18:55		47/468, 36/830	108/1714
	Ads	16.5	1,503,098	4,108,537	213,888	13:39		47/468, 36/830	94/1374
	MdsP		3,567,867	8,082,864	249,536	49:02			
	AdMsP		471,475	1,223,360	188,800	06:13			
	AdsP	8.8	407,018	105,4146	182,592	04:25			

Table 2: Verification Results for the Cache Module

always return the same time. BEG generated simple stubs for both the `Date` class and `System.currentTimeMillis()` method.

(4) Localization: The `java.util.ResourceBundle`, used for loading and reading error messages from a specified file location, was manually stubbed out. This class was not generated by BEG, since it was not referenced by any of the modules and did not have any effects on the module.

What application features enable BEG support? The more complex interactions between the module and the environment, the more complex analysis is needed to track them. Cohesive modules, simple data and control interactions, and absence of concurrency-related module-environment dependencies enable BEG support.

Here are some questions that point to future directions:

What are the limitations of the current approach and how can they be addressed in the future? The driver specification language could be extended to handle more Java expressions and statements, including support for property specification, e.g., assertions.

As for stub generation, we already mentioned that BEG does not calculate all possible module-environment interactions. Adding analyses that are used in slicing, which calculates all possible dependencies, would give more power and flexibility to BEG. Also, once the drivers are generated, off-the-shelf slicers (e.g., the Indus Java slicer[24]) can be used to reduce the resulting system. Preliminary results show that BEG environment generation is a more aggressive approach than slicing, however, at a cost of safety. Slicing, while a safe technique, does not scale as well and may retain too many dependencies in the final slice. We believe, we can combine both slicing and stub generation to strike the balance: first, stub generation can be used to abstract classes that definitely do not influence the property, then, slicing can be applied to the reduced system.

Native methods is another big problem. In many cases, we believe, *domain-specific* knowledge can be used to produce faithful stubs that can be reused across multiple analyses.

Another limitation of the current approach is treatment of TOP values, which can be safely interpreted by abstraction engine (e.g., in Bandera [9]) or executed symbolically (e.g., [20]). However, in this paper, we did not use automated support to treat the TOP values according to their semantics; we had to do it manually. This limitation can be addressed by using symbolic execution in JPF or using test case generation to refine TOP values.

What is the impact of POR on performance of environment generation? According to [12], the majority of methods in an arbitrary application (especially, library methods) are meant to be *atomic*. Atomicity property means that the outcome of a method execution does not depend on thread interleavings, i.e., for analysis purposes, it is sufficient to check only one, e.g., sequential, execution of an atomic method. With POR on, JPF executes many methods atomically. We thought that behavior added or excluded from atomic methods would not have great impact on the state space exploration, i.e., that POR would decrease reductions by BEG. While some examples in this case study exhibit this behavior, others do not. We can not generalize on the interaction between environment generation and POR until more case studies are carried out. It is possible that some particular features of the case study (or the BEG-generated environments) prevent consistent interaction between environment generation and POR. Regardless of how POR influences reductions by environment generation, the examples in this paper show that environment generation is capable of producing nontrivial reductions on top of POR.

Can we automate environment refinement? In this work, we manually went through several refinements of the module-environment system. For example, based on model checking

results, we adjusted modules to include classes that were tightly coupled with the module. Based on coverage information, we adjusted driver specifications (e.g., to include more calls to the module). Based on model checking results, we refined stubs (e.g., to include side-effects). It would be useful to have automated support for refinement of modules, drivers, and stubs based results of model checking and coverage information.

7. RELATED WORK

7.1 Testing

Environment generation presented in this paper is similar to unit testing. One of the most popular frameworks for Java unit testing is JUnit[19]. JUnit tests are designed to test a Java class in isolation. Mock objects[14] are used to fake environment components. In Java, there are tools (e.g., JTest [18]) that use simple structural analysis of the Java class under test to automatically generate JUnit tests. Compared to BEG-generated drivers, JTest-generated JUnit tests exercise Java classes in a limited way: in a sequential setting, with short method call sequences and bounded parameter values.

Driver generation is similar to test-case generation. Recent efforts in automated test-case generation rely on generating predicates from precondition specifications [5], but it is not known how they scale for large commercial client-server software.

7.2 Static Analysis

There are many examples of using static analysis techniques to aid modular verification. Verisoft incorporates a static analysis for closing partially-implemented systems by calculating branch conditions influenced by environment and interpreting such conditions as TOP [8]. Stoller [27] describes an approach that computes a partition of a system's inputs based on a dataflow analysis. Using a single representative input value from each partition, one can build a set of test cases that exercise all behaviors of the system, while avoiding redundancy. Rountev et al. [25] explore how points-to and side-effects analyses may be used to produce summaries for library modules that later can be used for separate analysis of client modules.

There is also work on generation of environment assumptions for optimistic environments [1, 7, 32]. This work is aimed at finding environments within which the module would satisfy its required properties. Given an implementation of an actual environment and the assumptions for the optimistic environments, one can check whether the implementation violates any assumptions. In our case, however, we do not have actual environments (we need to generate them) and we can not use optimistic environments because they do not uncover errors; they avoid them.

7.3 User Assumptions

BEG approach to environment generation from specifications builds on the work of Avrunin et al. [2] who developed tool support for analyzing partially implemented real-time systems with components implemented in Ada or specified using graphical interval logic and regular expressions. In [23], LTL was used to describe driver assumptions.

Another modular approach to checking multi-threaded programs is implemented in Calvin [13]. Their approach is

aimed at procedure checking and relies on user specifications of environment assumptions that describe other procedures in the system and constrain interactions among threads.

In some cases, the domain-specific knowledge can be used to automatically generate environment models directly without having a user write environment assumptions. Stoller et al. [28] show how a distributed (multi-process) Java program can be transformed into a single-process program and how native RMIs can be replaced with ordinary methods that simulate RMIs. While their paper does not present any case studies, we believe, the approach could be useful within BEG, especially if one wants to check properties that depend on preserving semantics of distributed programs.

The best approach to writing correct programs is to start with a correct design. For example, design for verification with concurrency controllers enables effective modular verification by decoupling the behavior of concurrency controllers from the behavior of threads that use them. In [4], Betin-Can et al. describe application of their technique to a large case study, which had to be manually reengineered to conform to the design. We believe, reengineering I-BPM would be a difficult task.

8. CONCLUSIONS AND FUTURE WORK

Environment generation is a problem persistent across different types of program analysis. The field of automated environment generation has not been extensively studied due to lack of automated support and case studies that show the value of application of automated environment generation to real software.

In this paper, we used Bandera Environment Generator (BEG) to generate environments for two modules belonging to a large commercial client-server Java application. Our experience has been very promising. BEG generated environments that:

- for faulty modules, uncovered the errors, yet,
- for error-free modules, covered as much module behavior as the previously written by hand environments.

While current BEG approach to environment generation is not safe, we have shown that it is a practical method for finding errors efficiently. The limitations can be addressed by incorporating more detailed specification and deeper static analyses approaches.

9. ACKNOWLEDGMENTS

We kindly thank Graham Hughes for the help with the manual environments; JPF and Bandera teams for the help with the tools; and the anonymous reviewers for their detailed comments.

10. REFERENCES

- [1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–109, New York, NY, USA, 2005. ACM Press.
- [2] G. S. Avrunin, J. C. Corbett, and L. Dillon. Analyzing partially-implemented real-time systems. In *Proceedings of the 19th International Conference on Software Engineering*, pages 228–238, 1997.

- [3] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 203–213, June 2001.
- [4] A. Betin-Can, T. Bultan, M. Lindvall, S. Topp, and B. Lux. Application of design for verification with concurrency controllers to air traffic control software. In *Proceedings of the 20th IEEE International Conference on Automated Software Engineering*, 2005.
- [5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates, 2002.
- [6] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
- [7] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS 2619)*, 2003.
- [8] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing open reactive programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357, 1998.
- [9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
- [10] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.
- [11] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser. Analyzing interaction orderings with model checking. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 154–163, Washington, DC, USA, 2004. IEEE Computer Society.
- [12] C. Flanagan and S. Freund. Atomizer: A dynamic atomcity checker for multithreaded programs. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, 2004.
- [13] C. Flanagan and S. Qadeer. Thread modular model checking. In *Model Checking Software (LNCS 2648)*, May 2003.
- [14] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. jmock: supporting responsibility-based design with mock objects. In *OOPSLA Companion*, pages 4–5, 2004.
- [15] A. Groce and W. Visser. Model checking java programs using structural heuristics. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 12–21, New York, NY, USA, 2002. ACM Press.
- [16] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 13(4), 2000.
- [17] G. Hughes, S. P. Rajan, T. Sidle, and K. Swenson. Error detection in concurrent java programs. In *Workshop on Software Model Checking*, 2005.
- [18] P. JTest. Website. <http://www.parasoft.com>.
- [19] JUnit. Website. <http://www.junit.org>.
- [20] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.
- [21] M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *The First Symposium on Networked Systems Design and Implementation*, 2004.
- [22] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS real-time scheduling kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
- [23] C. S. Păsăreanu. *Abstraction and Modular Reasoning for the Verification of Software*. PhD thesis, Kansas State University, 2001.
- [24] V. Ranganath. Indus Website. <http://indus.projects.cis.ksu.edu>.
- [25] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proceedings of the 10th International Conference on Compiler Construction (CC'01)*, 2001.
- [26] S. D. Stoller. Model-checking multi-threaded distributed java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 224–244, London, UK, 2000. Springer-Verlag.
- [27] S. D. Stoller. Domain partitioning for open reactive systems. In *Proceedings of the international symposium on Software testing and analysis*, pages 44–54. ACM Press, 2002.
- [28] S. D. Stoller and Y. A. Liu. Transformations for model checking distributed java programs. In *Proc. 8th Int'l. SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 192–199. Springer-Verlag, May 2001.
- [29] O. Tkachuk. Bandera Environment Generator Website. <http://beg.projects.cis.ksu.edu>.
- [30] O. Tkachuk, M. B. Dwyer, and C. S. Păsăreanu. Automated environment generation for software model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, Oct. 2003.
- [31] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering*, Sept. 2000.
- [32] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.
- [33] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, 2004.