DOMAIN-SPECIFIC ENVIRONMENT GENERATION FOR MODULAR SOFTWARE MODEL CHECKING

by

OKSANA TKACHUK

M.S, Kansas State University, 2003

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2008

Abstract

To analyze an open system, one needs to close it with a definition of its environment, i.e., its execution context. Environment modeling is a significant challenge: environment models should be general enough to permit analysis of large portions of a system's possible behaviors, yet sufficiently precise to enable cost-effective reasoning. This thesis presents the Bandera Environment Generator (BEG), a toolset that automates generation of environment models to provide a restricted form of modular model checking of Java programs, where the module's source code is the subject of analysis along with an abstract model of the environment's behavior.

Since the most general environments do not allow for tractable model checking, BEG has support for restricting the environment behavior based on domain-specific knowledge and assumptions about the environment behavior, which can be acquired from a variety of sources. When the environment code is not available, developers can encode their assumptions as an explicit formal specification. When the environment code is available, BEG employs static analyses to extract environment assumptions. Both specifications and static analyses can be tuned to reflect domain-specific knowledge, i.e., to describe domain-specific aspects of the environment behavior. Initially, BEG was implemented to handle general Java applications; later, it was extended to handle two specific domains: Graphical User Interfaces (GUI) implemented using the Swing/AWT libraries and web applications implemented using the J2EE framework. BEG was evaluated on several non-trivial case studies, including industrial applications from NASA, SUN, and Fujitsu. This thesis presents the domain-specific environment generation for GUI and web applications and describes BEG, its extensible architecture, usage, and how it can be extended to handle new domains.

DOMAIN-SPECIFIC ENVIRONMENT GENERATION FOR MODULAR SOFTWARE MODEL CHECKING

by

OKSANA TKACHUK

M.S., Kansas State University, 2003

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2008

Approved by:

Co-Major Professor
Matthew Dwyer

Approved by:

Co-Major Professor
John Hatcliff

Copyright

Abstract

To analyze an open system, one needs to close it with a definition of its environment, i.e., its execution context. Environment modeling is a significant challenge: environment models should be general enough to permit analysis of large portions of a system's possible behaviors, yet sufficiently precise to enable cost-effective reasoning. This thesis presents the Bandera Environment Generator (BEG), a toolset that automates generation of environment models to provide a restricted form of modular model checking of Java programs, where the module's source code is the subject of analysis along with an abstract model of the environment's behavior.

Since the most general environments do not allow for tractable model checking, BEG has support for restricting the environment behavior based on domain-specific knowledge and assumptions about the environment behavior, which can be acquired from a variety of sources. When the environment code is not available, developers can encode their assumptions as an explicit formal specification. When the environment code is available, BEG employs static analyses to extract environment assumptions. Both specifications and static analyses can be tuned to reflect domain-specific knowledge, i.e., to describe domain-specific aspects of the environment behavior. Initially, BEG was implemented to handle general Java applications; later, it was extended to handle two specific domains: Graphical User Interfaces (GUI) implemented using the Swing/AWT libraries and web applications implemented using the J2EE framework. BEG was evaluated on several non-trivial case studies, including industrial applications from NASA, SUN, and Fujitsu. This thesis presents the domain-specific environment generation for GUI and web applications and describes BEG, its extensible architecture, usage, and how it can be extended to handle new domains.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I want to thank the people whose support made this thesis possible. First of all, I wish to express my gratitude to my advisor, Matthew Dwyer, for giving me a challenging and interesting problem to work on, for his constant positive encouragement, for reading several rough drafts of this thesis, and providing constructive comments.

I am thankful to the professors at Kansas State University whose classes sparked my interest in research and continuation of my education. I am grateful to John Hatcliff, who graciously agreed to step into the co-advisor's shoes after Dr. Dwyer transferred to the University of Nebraska - Lincoln. I thank my committee – my advisors, Robby, Torben Amtoft, Steve Warren, and Medhat Morcos – for reading my thesis and providing helpful comments. Special thank you goes to Robby for his support with the Bogor model checking framework and Gurdip Singh for helping with the graduation paperwork.

I am grateful to the people behind the JPF model checking framework, especially Willem Visser, Corina Păsăreanu, and Peter Mehlitz, for their constant support with JPF.

I want to express my appreciation to my colleagues at Fujitsu Laboratories of America – Sreeranga Rajan, Indradeep Ghosh, Mukul Prasad, and Ryusuke Masuoka – for supporting my work on environment generation and for giving me a chance to apply my methodology to Fujitsu's applications.

I am grateful to my family and friends, especially Tima, Nadia, Marina, Vadim, Natasha, Max, Olya, Maxim, Kuzya, Misha, Umid, and Annabelle, for the wonderful summer camping trips, winter skiing trips, and dinner parties. They always gave me something to look forward to while I was working on my thesis. At last, I want to thank my father and my mother, who always encouraged me to pursue high education and gave me their love and support.

Dedicated

To My Beloved Mother,

Nadia Tkachuk

Chapter 1

Introduction

Research efforts [14, 35, 92, 97] show that model checking [11] can be an effective technique for detecting concurrency-related errors in software systems. However, due to scalability issues, to handle industrial-size software, model checking needs to be combined with powerful reduction techniques such as partial order reduction [77], data abstraction [23], predicate abstraction [5], slicing [22], heuristic search [36], or modular model checking [50]. In this thesis, we pursue the modular approach, which restricts analysis to selected parts of a program, called a *unit* under analysis.

Units are *open* systems, which may interact with other components, whereas model checking requires *closed*, i.e., complete, systems. To model check a unit in isolation, one has to close it with a model of its execution context, which we refer to as an *environment*. Given a Java program, we consider its decomposition into two parts: a unit under analysis and the unit's environment. The main idea behind this decomposition is to model the environment at a high level of abstraction, thus reducing the state space of the entire system. The unit's source code is the subject of verification along with an abstract model of the environment's externally observable behavior. The resulting abstracted model can be analyzed against unit properties by existing Java model checking frameworks such as Bogor [70, 71] and Java PathFinder (JPF) [9, 45].

Environment generation is a significant challenge, since an environment should be general enough to cover interesting unit behaviors and uncover errors, yet restrictive enough to

enable tractable model checking, without being overly restrictive, which may cause the analysis to miss important behaviors and mask errors. Experience shows that environment generation is often done by hand (e.g., [64]) or omitted. For example, in [59], while model checking the 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. As a result, model checking could not complete the search. In general, interactions between a unit and its environment can be complicated and difficult to analyze: the environment can influence the unit's *control* (e.g., by invoking the unit's methods) and *data* (e.g., by modifying the unit's data flowing into the environment). In Java, both data and control interactions can also happen through synchronization, exceptions, global references, parameters, and return values.

Environment generation is a problem persistent across different types of program analysis: in unit testing, one has to write test *drivers*, components that make calls to the unit, and *stubs*, simplified implementations of actual classes and methods called by the unit; 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.

When modeling drivers and stubs, there are three aspects of environment behavior one has to model: (1) control, usually represented by sequences of actions the environment may perform on the unit; (2) data, the values the environment may pass to the unit through arguments or return values; and (3) concurrency, usually described by the number of threads in the environment and synchronization used. There are several automated approaches that can be used to model certain aspects of drivers and stubs:

- Structural Analysis: In unit testing, there are tools (e.g., JTest [46], JCrasher [17], Randoop [61]) that use structural analysis of Java classes under test to automatically generate JUnit [47] tests. However, automatically generated JUnit tests are limited to sequential drivers that perform short sequences of method calls with sample or random

argument values.

- User Specifications: Specifications can be used to describe more complicated sequences of actions a driver may perform on the unit. This approach is used in *assume-guarantee* model checking [50, 56, 65], where the environment is restricted by user-provided specifications called *environment assumptions*. Various formalisms have been used for writing environment assumptions, e.g., Linear Temporal Logic (LTL) [56] in [66], Graphical Interval Logic (GIL) [21] and regular expressions in [4]. Most of these concentrate on describing sequences of method calls the environment may perform on the unit, yet, they do not address specification of synchronization and data values flowing from the environment to the unit.

- Symbolic Execution: There are tools that address generation of data values based on symbolic execution (e.g., Korat [8], Symstra [96], Kiasan/KUnit [20]). However, these tools work for sequential programs, do not scale for large units, and rely on user specifications such as method pre- and post-conditions.

- Static Analysis: Deeper static analyses can be employed to discover interesting environment sequences (e.g., sequences of methods that raise unit exceptions [94]) and to identify environment data partitioning (e.g., [78]). In general, such analyses can target only specific environment features. Slicing [38] can be used to calculate all possible dependencies between a unit and its environment. However, slicing can be over-approximate and usually requires complete programs, including stubs for native methods.

- Run-Time Analysis: Monitoring program executions allows one to learn patterns of environment behavior [3], yet, such techniques require that the environment is set up and the unit under analysis is running.

Section 2.2 presents more related work. In spite of many approaches that can be used for

environment generation, most only address specific aspects of program behavior and work for sequential programs.

In this thesis, we present the Bandera Environment Generator (BEG), a toolset for automated environment generation, which treats data and control dependencies between a unit and its environment. The BEG approach evolved based on experience applying it to many case studies. In addition to providing techniques to calculate various aspects of environment behavior, the BEG case studies reveal that certain domains require modeling of domain-specific features. We present the environment generation methodology for (1) general Java programs, (2) Java programs with Graphical User Interface (GUI), written using the AWT and Swing libraries, and (3) web applications written using the J2EE framework.

1.1 Problem Definition

Java software is usually built as a collection of classes or packages that may be implemented independently and integrated later to produce a desired system. In such a setting, it is natural to define a unit under analysis as a collection of Java classes. The environment is defined as a collection of classes with which the unit interacts. Interactions happen at the unit-environment *interface*, which is represented by unit and environment public methods and fields. Both a unit and its environment can perform *actions* at each other's interface. These actions can be assignments to public fields and invocations of public methods, which may return data, including exceptions.

Given a unit as a collection of Java classes, we wish to build a model of its environment. The modeled environment classes are broken into drivers and stubs. We define *drivers* as Java classes that hold a thread of control, i.e., classes containing the `main()` method or classes that extend/implement `java.lang.Thread/java.lang.Runnable`. The remaining environment classes are called *stubs*. In many cases, drivers exercise the unit behavior by performing sequences of actions on the unit, and stubs are the components that are used by the unit, e.g., library classes.

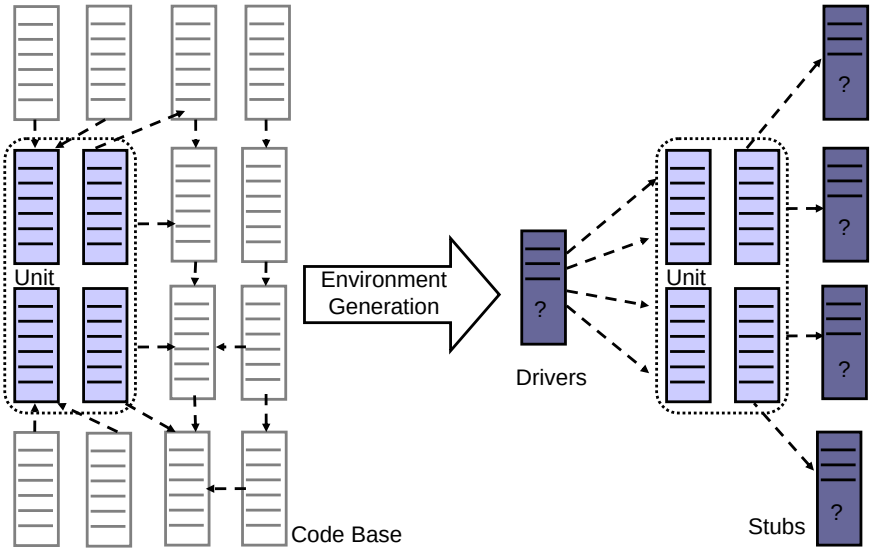

Figure 1.1: *Environment Generation Problem*

Since Java applications are built as a collection of classes that may be implemented independently and integrated later, there are two situations in which modular model checking is required: (1) the entire application is code-complete and represents a closed application that is too large for cost-effective analysis (in this case, the environment implementation is available) and (2) some environment classes are not code-complete or missing due to the open nature of a system. For example, GUI applications are inherently open and need a model of a GUI user before they can be treated as a closed system required for model checking. In this case, we may have documentation or other specification artifacts, e.g., a description of a screen transition diagram, use case scenarios, program usage, or behavior of missing components.

Figure 1.1 depicts the problem we want to solve. On the left, it shows a Java system represented by a collection of classes (represented by boxes containing lines of code). A unit is represented by a collection of classes enclosed with a dashed line; the rest of the classes in the system represent the environment. The arrows depict dependencies among classes;

arrows that cross the boundaries of the unit depict unit-environment interactions, which can be directed to and from the unit and can represent data or control dependencies. We want to generate code for the environment's *observable* behavior, i.e., code for drivers and stubs that directly interact with the unit, preserving the behavior of the entire environment that may affect the unit, while abstracting away behavior that is internal to the environment. The right part of Figure 1.1 illustrates the problem by indicating the result of environment generation with "?" on drivers and stubs. Note that the picture shows a common case when drivers make calls to the unit and stubs are called by the unit. In fact, all of the case studies performed to evaluate BEG fall into this category. In general, the interactions may be arbitrary (e.g., stubs may have callbacks to the unit and the unit may have callbacks to drivers).

1.2 Proposed Solution

Thorough treatment of the mechanisms by which the environment may influence the unit's behavior is essential for cost-effective reasoning. The unit-environment interactions may include control and data dependencies, including synchronization-related ones. There are also cases when even code-complete applications cannot be model checked without modeling additional components, e.g., GUI applications. For this reason, we believe that multiple sources of information should be combined to generate environment models that reflect a broad range of realistic environment behaviors and that capture control and data interaction between a unit and its environment. In addition, to better understand what types of support are needed, we perform a number of case studies to learn about common environment features that need to be modeled for cost-effective model checking.

We implement our solutions to the environment generation problem in the Bandera Environment Generator (BEG), which, given a unit under analysis, generates code for its environment. The unit closed with the generated environment is model checked against the unit's properties using existing Java model checking frameworks such as Bogor [71] and Java

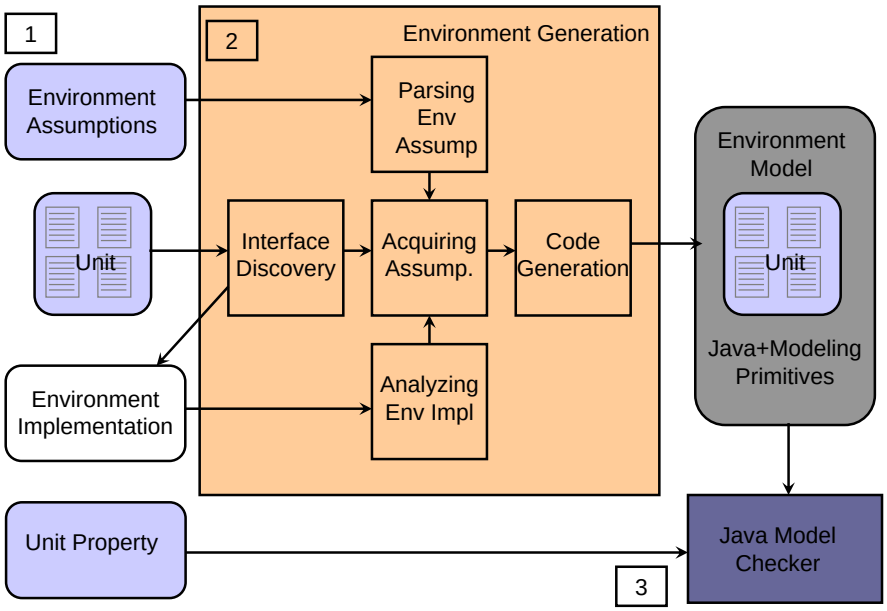

Figure 1.2: *Modular Model Checking Framework Using BEG*

PathFinder (JPF) [92].

Figure 1.2 shows the high-level architecture of our modular model checking framework, which consists of the following steps:

1. **Unit and Property Specification:**

   Users specify the unit under analysis as a collection of Java classes. They also specify the unit properties to be checked.

2. **Environment Generation:** Environment generation consists of the following three steps:

   (a) **Interface Discovery:** As the first step, BEG automatically discovers the structural information about classes, methods and fields in the unit-environment interface; we call it the *syntactic* interface. The behavior of the environment is discovered at the next step.

(b) **Acquiring Environment Assumptions:** Environment assumptions are used to describe the behavior of the environment. BEG supports acquiring environment assumptions from two sources: user specifications and environment code, if available. To extract environment assumptions from code, BEG employs static analysis.

In the absence of assumptions, BEG can be configured to generate *universal* drivers and stubs. Universal drivers consist of a number (specified in a configuration file) of unit class instances and threads. The methods of universal drivers and stubs exhibit the most general behavior (i.e., they perform any sequence of program actions that are exposed at the unit-environment interface).

(c) **Code Generation:** Using environment assumptions, BEG generates environment models encoded in Java using special *modeling primitives* to capture approximations produced by static analyses or defined by the user specifications. Generated environment code is integrated with the unit's code and passed to existing model checking frameworks.

3. **Model Checking:** Once the unit is closed with its environment, existing abstraction techniques can be used to further reduce the state space of the unit-environment system. The resulting abstracted program and the unit's properties are fed to existing Java model checking frameworks such as Bogor and JPF. Given a complete system, these model checkers exhaustively explore all possible paths in the system and, if a property violation is found, record the violating trace, called a *counterexample*.

Next, we describe the above steps in greater detail. Note that BEG supports environment generation steps, whereas the first step, unit and property specification, is done manually and the third step, model checking, is done using existing Java model checking frameworks.

1.2.1 Unit and Property Specification

In general, selection of the classes in the unit is driven by the properties that one wants to reason about. One can also use domain-specific knowledge. For example, for applications written using a framework, e.g., Swing/AWT or J2EE, the application-specific code is treated as the unit, whereas the framework code is treated as the environment.

To produce effective environments, it is better to choose a cohesive unit with classes tightly coupled internally but loosely coupled with environment classes. This strategy minimizes the number of unit-environment dependencies, which may be difficult to describe at a specification level or difficult to automatically extract using static analysis.

1.2.2 Environment Generation

In this section, we describe the three steps of environment generation in BEG.

Interface Discovery

The unit-environment interface consists of two parts: (1) the unit interface, consisting of unit methods and fields that can be exercised by the environment, and (2) the environment interface, consisting of external classes, methods and fields referenced by the unit. Given a unit under analysis as a collection of Java classes, BEG automatically discovers both the unit and environment interfaces.

Information about methods and fields in the unit interface is used to construct program actions, i.e., method calls and field assignments, the environment may perform on the unit. These actions define an *alphabet* of environment actions that can be used to construct universal environments. These actions are also used to write user assumptions or check their validity.

The environment interface, i.e., classes, methods and fields referenced by the unit, gives information about the structure of stubs that need to be generated. The syntactic environment interface constrains the number of classes, methods, and fields that need to be

generated in the environment to those that directly interact with the unit.

Acquiring Environment Assumptions

If the implementation of environment classes is available, then static analyses can be used to discover certain environment assumptions. If some environment classes are missing, e.g., the representation of a user for GUI applications, BEG supports specification of different aspects of environment behavior. BEG has the following support for specifying user assumptions and extracting environment assumptions using static analyses:

- **Specifying Assumptions:** BEG supports generation of drivers and stubs from user specifications. Such drivers and stubs can instantiate unit classes and perform sequences of actions on those instances. BEG allows users to specify compactly the number of threads in the driver, the number of instantiations, and the sequences of actions performed by drivers and stubs using LTL and regular expressions, using method invocation and assignment expressions as atomic actions.

  To specify data values (e.g., arguments for method invocation), BEG has the following support. If exact data values are known, then *concrete* data values can be specified. If concrete values are unknown, then *choice* or *abstract* values can be used. Choice values encode a nondeterministic choice over a range of values of the specified type. Abstract values are used to denote *any* value of the appropriate type. In addition, when specifying a method call, one can omit values for receiver objects and parameter values; BEG fills the holes with values of the appropriate types.

- **Extracting Assumptions:** Various static analyses can be used to extract control, data and concurrency-related effects. Currently, BEG supports *side-effects* analysis [87, 85] to capture the way the environment may modify unit data. Additionally, side-effects analysis can be tuned to track side-effects to specific fields belonging to the unit or environment. We developed several variants of side-effects analysis to track specific features of Swing/AWT and J2EE components. For example, we identified

containment as an important property for environment components, useful for both GUI and web applications. Containment analysis tracks side-effects to fields which may store unit data. Additionally, we identified features like *visibility* and *enabledness* that are important for GUI components. We describe more domain-specific features in sections 4.1 and 4.2. BEG analyses are adapted specifically for environment generation problem: they are modular, compositional, parameterized, and produce summaries for environment methods.

Environment Code Generation

BEG translates environment assumptions into Java code. For example, given assumptions in LTL, BEG translates them into an automaton that is encoded as Java code. Special *modeling primitives* are used to reflect *nondeterminism* in the environment introduced by the user specifications or produced by the *approximation* of static analyses. For example, to reflect the *possibility* of an environment action, it is encoded as `if(Verify.randomBool()){action}`, which forces an underlying model checker to explore both branches of the `if` statement: the one where the action happens and the one where it does not.

When generating action code, each atomic action is translated into a legal Java assignment or method invocation, including receiver objects and argument values. Choice values are represented by a nondeterministic choice over a set of values. For example, `Verify.randomObject(String type)` is interpreted as a nondeterministic choice over a set of heap objects of the specified type in the program state where the call is executed. Abstract values represent any value of the corresponding type; they are encoded using `Abstraction.TOP` modeling primitives. Both Bogor and JPF, used in this work, support the modeling primitives used in our environment generation framework and they are natural to implement in any explicit-state model checker.

1.2.3 Model Checking

Model checking results can only be as good as environment models. For example, if the environment misses some important unit behavior, then "verified" results cannot be trusted. Also, if an error is found but it is introduced by an infeasible environment path, then the error is spurious. Therefore, we need to evaluate the quality of environment models produced by BEG. Since it is not always feasible to model check a unit closed with the universal environment, we use *fault detection* and *coverage* to evaluate the quality of environment models: a high quality environment model will uncover a property violation or, in case of no violations, will produce high coverage for the unit under analysis. Several metrics can be used to evaluate coverage during model checking [72]. In this thesis, we use *branch* coverage over the unit code to evaluate quality of generated environments.

The model checking step may uncover that the environment model needs *refinement*, i.e., it needs to be constrained in case of intractable model checking or it needs to be expanded in case of poor coverage. In this work, the refinement step is done manually, e.g., the user may need to refine environment assumptions and regenerate the environment code or manually edit stubs, generated based on static analysis results. Therefore, while BEG offers automated support for environment generation based on user specifications and static analysis results, the desired environment model may need manual refinement.

1.3 Thesis Contributions

We envision two ways in which BEG can be used effectively: during component development as an adjunct to traditional unit testing approaches and during program validation to enable more efficient reasoning and to model non-source-code components.

During component development, individual classes, or groups of classes, that constitute cohesive functional components, perhaps structured as Java packages, may become code complete when the code they interact with (e.g., client code) has not been written. In this setting, the classes form a unit and the missing classes they interact with form the

environment. To enable effective checking, we expect that developers will need to encode assumptions about the behavior of the environment at the unit's interface to account for both control and data effects. These assumptions can subsequently be checked against implementations of the missing environment classes as they become code complete.

During program validation, when considering a complete application, one may break the system into parts to enable more efficient checking of program properties. In this setting, the user selects classes that define the unit under analysis and the environment model is automatically extracted. For applications that interact with external entities, such as embedded control software processing data from hardware devices, developers may incorporate assumptions about those interactions to generate a representative model of the external environment.

The key issue in using our environment generation techniques is balancing the human cost (i.e., the effort involved in writing specifications), the tool cost (i.e., the expense of verification), and the degree of coverage (i.e., the coverage of unit code). While environment generation in general is a challenging and difficult to automate problem, by focusing on specific domains and features, we make it practical. While some of our techniques require manual effort, e.g., environment refinement, the models produced for specific domains are usually precise and reusable across multiple applications from the same domain. For example, stubs developed for Swing/AWT and J2EE libraries are reusable, so the cost of their development can be amortized.

Our approach builds on existing work in unit testing, assume-guarantee reasoning, and static analysis. This thesis makes the following contributions:

- **Automated interface discovery:** We provide support for automatic discovery of the syntactic unit-environment interface. This information is used to build universal environments, to check the validity of user assumptions, and to constrain the environment model to classes, methods, and fields that directly interact with the unit.

- **Acquiring environment assumptions from a variety of sources:** We adapt

several existing specification forms such as LTL and regular expressions to describe environment behavior. While LTL and regular expressions can be used to describe patterns of program actions, BEG specification language also allows for specification of data and concurrency-related aspects of environment behavior. We also adapt existing static analyses techniques to work specifically for environment generation problem. Our analyses are modular, compositional and parameterized.

- **Environment code generation:** We describe strategies for encoding environment assumptions into Java code that can be processed by Java abstraction and model checking tools.

- **BEG implementation:** We implement a tool parameterized by different sources and descriptions for environment assumptions. The tool has an *extensible architecture*, so that new strategies for environment assumptions and code generation can be easily added.

- **Extensive case studies:** We evaluate our approach on a number of non-trivial Java examples, taken from different domains: GUI applications and Java applets, written using Swing/AWT libraries, and web applications written using J2EE libraries. To show the scalability of our approach, our studies include real industrial applications: NASA's Autopilot [74], Fujitsu's I-BPM and SUN's Pet Store [82]. The Autopilot and I-BPM case studies also demonstrate that BEG-generated environments are more cost-effective than previously written manual environments.

- **Environment generation methodology:** Based on the case studies, we develop and describe a methodology that can be used to perform environment generation for general Java applications, as well as domain-specific strategies for GUI and J2EE applications.

1.4 Thesis Organization

The next chapter gives background and related work on unit testing, static analysis, and modular model checking. Chapter 3 describes our general methodology and illustrates it on a small publish-subscribe example. Chapter 4 describes domain-specific methodologies developed for GUI and J2EE applications. Chapter 5 describes BEG's techniques for unit-environment interface discovery, specifying and extracting assumptions, and code generation. Chapter 6 describes BEG's extensible architecture, options and configurations. Chapter 7 discusses case studies, including industrial applications. We conclude and describe future work in Chapter 8.

Some of the material presented in this thesis has been published in the form of articles. Our environment generation approach for GUI applications, described in section 4.1, is based on [26]. Presentation of BEG specification language in section 5.3 is based on [88]. Presentation of static analysis in section 5.4 is based on our earlier work [85, 87]. We extend our previous static analysis to work specifically for GUI and J2EE applications. Some experiments in Chapter 7 have been described in [26, 86, 88, 89].

Chapter 2

Background and Related Work

In this Chapter, we presents background and related work on techniques related to this thesis, specifically, unit testing, static analysis, and model checking. For each approach, we describe general techniques and some tools that implement them for analysis of Java programs. As we demonstrate, each approach has to confront the environment generation problem, i.e., building a model of the environment for a system under analysis.

2.1 Background

2.1.1 Unit Testing

There are many testing techniques (e.g., black box testing, white box testing). Our techniques are most similar to unit testing. To test a unit, developers need to provide a framework that allows for exercising the behavior of the unit. A testing framework includes a program that simulates the behavior of the environment by passing sequences of inputs or test cases to the unit, executing the code of the unit and recording the outputs. This program is called a *test harness* or a *test driver* (this is similar to environment drivers in our framework). To simulate complex or missing environment components, e.g., a database, mock objects [32] are frequently used (this is similar to environment stubs in our framework). For easy maintenance, a test harness takes test cases from a repository of tests, feeds them to the unit, and saves the results into a repository for later examination. Correctness

of outputs is identified by *oracles*, which examine test executions and record those that do not match the expected results (this is similar to the model checking step in our framework). Additionally, coverage is used to evaluate how extensively a test suite exercises the unit's behavior. The types of coverage include statement, branch, condition, and path coverage [98].

Overall, unit testing is a widely used technique that scales well and can be evaluated based on coverage results. However, unit testing requires generation of test cases (i.e., the environment) that produce high coverage, avoiding redundancy. This work usually involves a lot of manual labor and is mostly targeted at sequential code.

2.1.2 Static Analysis

Static analysis is a compile-time technique used to collect information about program run-time behavior. A well-studied application of static analysis is code optimization, for example, dead code elimination, improvement of registers usage, and elimination of redundant computation. Another application is detecting potential errors or proving interesting properties of programs, for example, finding null pointer dereferences or proving their absence.

In this thesis, we adapt *side-effects* analysis [51], which determines a set of memory locations modified by a program statement or method. This analysis requires information from *points-to* analysis [55], which determines the objects a reference variable may point to. Another related analysis is *slicing* [84], which given a set of program points (e.g., program statements), called a *slicing criteria*, automatically calculates the program points that may influence the execution of the slicing criteria. Slicing calculates all possible data and control dependencies between the slicing criteria and the rest of the program points.

Static analysis used for code optimization and program verification must be *safe*, i.e., the information it collects should be true for all possible program executions and inputs; to be computable, due to fundamental undecidability theory, the results are usually approximate.

When used to prove correctness of programs or to detect errors, static analysis is comple-

$$
\begin{aligned}
Data_{exit}(s) &= (Data_{entry}(s) - Kill(s)) \cup Gen(s) \\
Data_{entry}(s) &= \bigsqcup \{Data_{exit}(s') \mid s' \in pred(s)\} \\
Data_{entry}(s_0) &= Init
\end{aligned}
$$

Figure 2.1: *Data Flow Equations for Forward Data Flow Analysis*

mentary to traditional testing techniques and model checking. Due to analysis imprecision, static analysis is less precise but, in general, more scalable than model checking. Due to safety, unlike traditional testing techniques, static analysis can produce formal correctness results.

There are several approaches to static analysis, among them are Data Flow Analysis (DFA) [60] and Abstract Interpretation [16]. Such approaches require the analysis designer to decide on a cost-precision tradeoff: the more precise the analysis, the more costly. In this thesis, we employ DFA to calculate specific unit-environment dependencies.

2.1.3 Data Flow Analysis

In DFA frameworks, a program is represented as a Control Flow Graph (CFG), where nodes represent basic blocks of the program, e.g., sequences of statements, and edges represent control flow between them. The basic approach of DFA is to propagate abstract facts through CFG nodes. For example, points-to analysis propagates points-to graphs or reference variables and sets of objects they may point to through CFG nodes and side-effects analysis propagates sets of modified objects through CFG nodes.

When propagating abstract facts through CFG nodes, one specifies the direction of propagation: *forward* analyses propagate data along the control flow (e.g., points-to and side-effects analyses); *backward* analyses propagate data in the direction opposite to the control flow (e.g., live variable analysis, which calculates for each program point the variables that may be potentially read before their next write). For each CFG node, e.g., a program statement s, DFA calculates both its incoming and outgoing data, $Data_{entry}(s)$

and $Data_{exit}(s)$. The incoming information flowing though a CFG node gets transformed by a *transfer function* according to the semantics of the program statement at the node. Figure 2.1 shows a common form of transfer functions used in the equational approach for forward DFA. The first transfer function shows that a statement s generates data defined by the set $Gen(s)$ and overwrites data defined by the set $Kill(s)$. Information flowing out of a node is propagated to all of its successors, $succ(s)$. If a node has several incoming edges, to calculate its incoming data, the facts flowing from all of its predecessors, $pred(s)$, are combined according to a *combination operator*, $\bigsqcup$, which is commonly either intersection, $\bigcap$, or union, $\bigcup$. Analyses that use intersection calculate data that persists on *all* execution paths; such analyses are called *must* analyses. In contrast, *may* analyses use union as a combination operator and calculate data that holds on *at least one* execution path.

In addition to transfer functions, the incoming data for the initial node, s_0, are specified as a special initial value, $Init$. To analyze a program, the analysis first initializes the data at the entry point of the program, then repeatedly processes CFG nodes to calculate their incoming and outgoing data until a *fixed point* is reached, i.e, when repeated processing of statements produces no change in the abstract facts calculated for each CFG node. To guarantee the existence and reachability of the fixed point solution, the transfer functions and the data values have to satisfy certain conditions. A sufficient condition that guarantees a fixed point will be reached is that data values form a complete lattice and the transfer functions are monotone.

There are several features of DFA analyses that influence the degree of approximation of the analysis. Analyses that calculate data within a procedure without exploiting information about its caller or callees are termed *intraprocedural*; such analyses must account for other procedures pessimistically. Analyses that propagate facts across procedure boundaries are called *interprocedural*. *Flow-sensitive* analyses take into account the order of statements; less precise but faster, *flow-insensitive* analyses ignore the order of statements. *Context-sensitive* analyses distinguish between calls to the same procedure at different program points by

keeping track of context information at each call site; *context-insensitive* analyses ignore the context information. *Thread-sensitive* analyses take into account possible switches in control flow due to thread interleavings; *thread-insensitive* analyses ignore such interleavings. In general, insensitive analyses scale better, however, they produce less precise results. It is up to analysis designers to pick analysis features according to how they are willing to trade off precision for scalability.

2.1.4 Software Model Checking

Model checking [11] is an automatic technique to verify properties of models represented by finite-state transition systems. The process of model checking consists of modeling a system under analysis, specifying its properties, and finally verifying that the property holds for the model. If a violation of the property is found, model checking produces a *counterexample*, a violating trace, which can be used to pinpoint the source of the error.

Initially applied to verification of high-level designs, model checking has proven to be useful for verification of software systems, which in general are not finite-state. First, a program under analysis is modeled in a model checker's input language and program properties are stated in a formalism accepted by the model checker. For example, the SPIN model checker [41] accepts models written in Promela and properties expressed in Linear Temporal Logic (LTL) [56]. Since real software is not always finite-state, modeling may require the application of techniques to produce a finite-state model that overapproximates the executable behavior of the program.

Given a finite-state transition system and a property expressed in a logical formalism, model checking performs verification through exhaustive exploration of all the states reachable by the system and checking whether the property holds in each state. If a model is not finite-state, *bounded* model checking can be used to explore all possible paths of bounded length. Due to possible overapproximation of the model, the model checker may report a *false alarm*, a violating trace that exists in the model but does not exist in the actual

program.

The differences in model checking algorithms come from the order the states are visited and how the states are enumerated and stored. There are *explicit state* model checkers (e.g., SPIN, Bogor, JPF), which manipulate and store states explicitly (e.g., as bit-vectors) and *implicit state* model checkers (e.g., SMV [57]), which represent and manipulate symbolic encodings of sets of states (e.g., Binary Decision Diagrams (BDDs)).

Software model checking can identify concurrency-related errors such as deadlocks and race conditions, which are difficult to identify using traditional testing approaches. However, application of model checking to real software has been limited by the *state space explosion* problem, which states that the size of the model's state space grows exponentially with the number of independent components (e.g., threads). There are several solutions that address this problem; among them are:

- *Partial Order Reduction:* Partial Order Reduction (POR) [77] exploits commutativity of concurrently executed independent transitions, which result in the same state regardless of their order. Only one interleaving of independent transitions needs to be explored by a model checker.

- *Atomicity:* A procedure or code block is *atomic* if, for every arbitrarily interleaved program execution, the overall behavior and outcome of the execution is the same as if it was executed without any interleavings (i.e., in a single atomic step). Atomicity information (e.g., provided as user annotations) guides a model checker to avoid exploring all possible interleavings for an atomic block.

- *Abstraction:* Abstraction is a technique based on the theory of abstract interpretation [16]. Abstract interpretation calculates approximated program semantics from the concrete one by replacing the concrete domain of computation and its concrete semantic operations with, respectively, an abstract domain and corresponding abstract semantic operations. For instance, *data abstraction* is used to substitute a program's

concrete data values that have large domains with abstract representations that have small value domains. This works well when a property of the system depends on properties of the variables rather than on their concrete values.

- *Heuristic Search:* Various heuristics can be used to guide the search. For example, exploring states with blocked threads first to get to a deadlock [36].

- *Slicing:* Given a property, the slicing (mentioned in section 2.1.2) performs various dependency analyses to calculate the parts of the program that have no influence on the property; such parts can be safely sliced away [22].

Another solution is modular model checking, which we pursue in this work and describe next.

2.1.5 Modular Model Checking

The motivation for modular verification [37, 50] is that breaking the system into smaller components and analyzing them one at a time at the level of their interface behavior is cheaper than analyzing the whole system at the lower level of detail. There are several flavors of modular verification. In this thesis, we adapt the *assume-guarantee* [65] approach, which decomposes the system into two parts: a unit under analysis and its environment. The assume-guarantee specification has two parts: *guaranteed* behavior of the unit, which describes the unit's desired behavior, and *assumed* behavior of the unit's environment. An assume-guarantee specification expressed in LTL is a pair $\langle \phi, \psi \rangle$, where ϕ and ψ are both LTL specifications. The assume-guarantee approach is used to show that the behavior of the unit is guaranteed to satisfy ψ, assuming that the behavior of the environment satisfies ϕ. In case of LTL specifications, the pair $\langle \phi, \psi \rangle$ can be combined to a single LTL formula $\phi \implies \psi$. In this case, there are two approaches to assume-guarantee model checking compared in [66]: (1) model checking of the module closed with a universal environment against a specification of the form $\phi \implies \psi$ and (2) if ϕ is a safety property of the environment,

```
1  package gov.nasa.jpf.jvm;
2  public class Verify {
3    static public void assert(boolean cond) {
4          if (!cond) throw new RuntimeException("assertion␣failed");
5    }
6    ...
7    static public void beginAtomic() {}
8    static public void endAtomic() {}
9    static public int random(int max) { return 0;}
10   static public boolean randomBool() { return false;}
11   //extensions for environment generation
12   static public Object randomObject(String type){return null;}
13   static public Object randomReachable(String type, Object obj){return null;}
14 }
```

Figure 2.2: *JPF Modeling Primitives*

encoding ϕ directly into the implementation of the environment and model checking the module closed with the environment against the module specification ψ. Our framework allows for modular verification of open Java systems using both approaches. In addition, our framework provides flexibility for the second approach by allowing synthesis of Java environments using additional notations (e.g., regular expressions, side-effects descriptions).

2.1.6 Java Model Checking Frameworks

In this thesis, we use Java PathFinder (JPF) [9, 45, 92] and Bogor [70, 71], which we describe next.

JPF is an explicit state Java model checker built on top of a customized virtual machine that executes a Java program along all possible paths, checking for runtime errors (e.g., unhandled exceptions) and synchronization-related problems (e.g., deadlocks and race conditions). When running a program, JPF systematically explores all possible thread interleavings and inputs. JPF runs the bytecode directly, bypassing the step of modeling the system. To curb the state space explosion problem, JPF employs heuristic search and partial order reductions. JPF is an open source project and has been successfully used to check real industrial-size programs, e.g., DEOS [64] and an Air Traffic Control System [7].

Bogor is an explicit state model checker, designed to be extensible and customizable.

```
1  package edu.ksu.cis.bandera;
2  public class Abstraction{
3    public static int TOP_INT = 0;
4    public static boolean TOP_BOOL = false;
5    public static String TOP_STRING = "top";
6      ...
7  }
8  public class Bandera {
9    static public void beginAtomic() {}
10   static public void endAtomic() {}
11   static public int chooseInt(int max) { return 0;}
12   static public boolean choose() { return false;}
13   //extensions for environment generation
14   static public Object chooseClass(String type){return null;}
15   static public Object chooseReachable(String type, Object obj){return null;}
16 }
```

Figure 2.3: *Bandera Modeling Primitives*

Bogor's input language, Bandera Intermediate Representation (BIR), is designed to be extensible with new semantic primitives; Bogor's plug-in architecture allows adding new state space storage and exploration strategies. Bogor has been used to model check various domain-specific programs, including Java.

Both JPF and Bogor can be combined with the Indus slicer [68], which can help reduce the state space [22]. However, JPF, Bogor, and the Indus slicer require a closed program written in pure Java. All case studies presented in this thesis are open systems, which require environment generation before any whole-program analyses such as slicing or model checking can be applied to them.

Both JPF and Bogor support modeling primitives that denote nondeterministic choices used in this thesis. Given a set of values and a nondeterministic choice over that set, the underlying model checker systematically explores all values in the given set. Figures 2.2 and 2.3 shows excerpts from the `Verify` and `Bandera` classes, used in JPF and Bogor respectively to model nondeterministic choices. For example, `randomBool()` is a choice between $\{true, false\}$ and `randomInt(n)` is a choice over $\{0, \ldots, n-1\}$ [23]. The following methods were added to support our environment generation approach: `randomClass("C")`

is a choice over the allocated instances of class `C` in the program state where the call is executed and `randomReachable("C", obj)` is a nondeterministic choice over the allocated instances of `C` that are also reachable from `obj`.

The `Abstraction` class, shown in Figure 2.3, declares `TOP` fields for scalar types and for commonly used non-scalar types, e.g., `String`. The `TOP` values are used during environment code generation to emit unknown values. The `TOP` values denote *any* value of the specified type. Abstraction (e.g., [23]) and symbolic execution tools (e.g., [19]) can be configured to recognize `TOP` values and treat them according to their semantics.

2.2 Related Work

In this section we present approaches used in unit testing, static analysis, and modular model checking that are related to BEG techniques.

2.2.1 Unit Testing

Java unit testing tools, e.g., JUnit [47], provide a standard infrastructure for setting up test suites. Once a test suite is set up, it can be automatically run every time the code base changes. JUnit encourages developers to write unit tests, although, writing test cases and inserting checks that act as oracles is usually a manual process.

There are tools that automate the task of writing JUnit tests, e.g., Parasoft's Jtest [46], JCrasher [17]. These tools analyze the structure of Java classes under test, then generate and execute JUnit-format test cases designed to expose uncaught runtime exceptions and verify requirements expressed with Design by Contract annotations. While these tools automate the task of building JUnit test suites, they do so in an ad-hoc manner, resulting in test suites that may miss test cases or contain redundant tests. Tools like Jtest are valuable tools for setting up a JUnit test suite, yet, compared to BEG environments, Jtest-like tools exercise Java classes in a limited way: a single Java class is exercised in a sequential environment, the driver method call sequences are short, and parameter values are limited to corner cases

(e.g., `null` values, which are used to expose `NullPointerExceptions`).

Some tools address redundancy of JUnit tests, e.g, Randoop [61], however, these tools still address sequential code only and rely on generation of random values.

Other tools address generation of data values based on symbolic execution (e.g., Korat [8], Symstra [96], Kiasan/KUnit [20]). These tools work for sequential programs, do not scale for large components, and exploit user specifications such as method pre- and post-conditions (e.g., expressed in Java Modeling Language (JML) [52]).

2.2.2 Static Analysis for Java

Examples of Java static analysis tools and approaches range from analyzers that use shallow intraprocedural data flow analyses (e.g., FindBugs [27] can detect possible null pointer dereferences) to tools that use some interprocedural analyses (e.g., JLint [44] can detect deadlocks by building a global lock graph and checking that there are no cycles in it) to analyzers that perform a whole-program analysis to calculate all possible program control and data dependencies (e.g., Indus [68]).

Static analysis can infer program properties or mine specifications. Weimer et al. present a miner [93] that produces simple policies dealing with resource leaks and forgotten obligations. Specifically, the miner learns simple two-state FSM policies given by the regular expression (ab)* (e.g., an opened file should be closed). Whaley et al. [94] present a static technique that, for a given Java class, detects illegal sequences of two method invocations on an instance of the class. Specifically, if the first method sets a field of the class to a value that leads to an exception when executing the second method, then such a sequence is illegal. These techniques can be used to learn some patterns of actions the environment may perform on a unit, yet, these patterns are simple.

Stoller [78] describes an approach that computes a partition of a system's inputs based on the data-flow analysis of the system. The idea is to use a single representative input value from each partition to exercise all behaviors of the system and to avoid exercising the same

behavior twice. In contrast, BEG generates environment values based on user specifications or it fills missing values using abstract or choice modeling primitives. BEG approach relies on subsequent symbolic execution or abstraction phases to partition abstract values prior to model checking.

In Java, static analysis can be complicated by the extensive set of Java libraries that contribute to the size of the program under analysis and native code written in other languages. These generally require generation of stubs.

Another complication, for interprocedural analyses, is the need to resolve virtual method invocations. Java is an object-oriented language, where a call site usually has a form $r.m(a_1, ..., a_n)$ with r being a receiver object. At run-time, the invoked method is determined by examining the actual type of the receiver object and its superclasses and finding the first method that matches the signature of the called method. At compile-time, the actual type of the receiver object can only be estimated. One may employ a Class Hierarchy Analysis (CHA) to determine possible types of the receiver object based on its declared type; given a long inheritance chain, the results of CHA analysis may be too imprecise. More precise but costly analyses can be used for virtual invoke resolution, e.g., points-to analysis.

2.2.3 Points-to and Side-Effects Analysis

There are various implementations of points-to and side-effects analysis for Java.

Soot [69] and Indus [68] provide both analyses, however, these tools perform whole-program analysis, i.e., to produce safe results, the program under analysis needs to have `main` method and pure Java implementation for native methods. Such tools cannot analyze GUI or J2EE applications without performing environment generation first.

Rountev et al. [73] explore how points-to and side-effects analyses may be used to produce *summaries* for library modules that later may be used for separate analysis of client modules. Unlike in BEG, their summaries are produced using whole program analysis

under the worst-case assumptions about a client and are targeted at the optimizations of the client.

BEG analyses are customized to work specifically for the environment generation problem: they are modular [10, 95], parameterized [55], flow-sensitive [51], and produce method summaries that can be encoded in Java. BEG analyses are also extensible and can be tuned to track side-effects to specific objects. We extended BEG points-to and side-effects analysis to track containers, i.e., objects that can hold unit data, and to track data specific to GUI and J2EE libraries.

2.2.4 Modular Model Checking

Our approach to environment generation from specifications builds on the work of Avrunin et al. [4], who developed tool support to analyze partially implemented real-time systems whose components were implemented in Ada or specified using graphical interval logic and regular expressions. Our work also builds on the approach for model checking of partial software systems in Ada presented in [24, 25, 66] by Pasareanu et al., who used SPIN and SMV model checkers to verify safety properties of units closed with universal environments and environments synthesized from LTL assumptions. While these approaches concentrate on describing sequences of method calls the environment may perform on a unit, they do not address specification of data values flowing from environment to the unit.

Another example of modular verification is described in [13] and incorporated into the Verisoft model checker [35]. Colby et al. use static analysis to close an open system by calculating the influence of externally defined data. The main idea behind their analysis is to find all conditionals dependent on external data and substitute them with a nondeterministic choice. Unlike in our approach, they use a simple notion of data dependence to drive their analysis, which has no ability to control the precision of the generated environment.

A modular approach to checking multi-threaded programs is implemented in Calvin [30]. Their approach is aimed at procedure checking and relies on user specifications of environ-

ment assumptions that describe other procedures in the system and constrain interactions among threads. Unlike in our framework, theirs allows for simple invariant specifications and requires that programs obey a restricted class of locking disciplines with respect to thread interactions.

2.2.5 Java Model Checking

Model checking of Java programs faces the same challenges as Java static analysis, only to a greater extent, since model checking can be viewed as a most precise, least scalable form of static analysis. Extensive Java libraries and infinite data domains contribute to the state space explosion problem. Native code cannot be handled and needs to be modeled first, either using pure Java or a modeling language of the underlying model checker. This means that programs with file IO or Remote Method Invocation (RMI) need to be preprocessed first to model the parts of the application that make use of native code. In addition, open interactive systems (e.g., GUIs and web applications) need a model of the user to appropriately close the system.

Both Bogor and JPF have been used to perform model checking of general Java programs as well as domain-specific programs [6, 42, 63]. For some approaches [6], a collection of models for specific library classes is produced. The advantage of such an approach is that once the models are generated, they can be reused across multiple analyses from the same domain. The limitation of the approach is its applicability to a specific type of applications. Next we describe several domains where Java model checking has been applied.

Stoller et al. [79] show how a distributed (multi-process) Java program can be transformed into a single-process program using three automated transformations: (1) centralization: combining multiple processes into a single process; (2) RMI removal: replacing of native RMIs with ordinary methods that simulate RMIs; (3) Pseudo-crypto: replacing native cryptographic operations with their simulations. Unlike our work, this approach has been carried out mostly at the theoretical level, with no real large case studies.

Another approach to model checking distributed Java applications is implemented in NetStub [6], a framework that contains reusable stubs for `java.net` and `java.nio` libraries. This approach is similar to our approach of developing reusable domain-specific stubs. Unlike in our work, their approach has no automated support for stub generation.

There are related approaches to specification and generation of environment for model checking of software components [42, 63]. The approach by Parizek et al. [63] works for software components with well-defined behavioral specifications written in ADL. Given such components, they derive drivers by calculating the inverse of ADL component specifications. In their framework, specifications already exist and, due to the nature of ADL components, which make no external calls, there is no need to generate stubs. Their approach addresses a specific instance of the environment generation problem, whereas BEG addresses a more general problem and has automated support for interface discovery and stub generation.

The approach by Hughes et al. [42], called interface grammars, is based on using Context Free Grammar (CFG) to describe usage of Java components. CFG is more expressive than automata-based specifications, e.g., LTL or regular expressions used in our work, however, this approach incorporates user specifications only. Using their approach, one can specify API usage of a Java class and generate a stub for it, which will check that the unit under analysis uses the class according to the specified grammar. While the CFG approach is more expressive, it does not scale to large systems, as only one stub at a time can be described by the interface grammars. Our case studies show that stub generation step can produce thousands of stubbed out classes; to perform stub generation on such a scale, one needs automated support, e.g., static analysis in BEG.

Chapter 3

Overview

In this chapter, we demonstrate our approach on a small publish-subscribe program, which illustrates the use of the observer pattern [33]. Then, in section 3.2, we describe our environment generation methodology shaped by several large case studies and different domains.

3.1 Example: Observer-Observable

In this section, we illustrate how to use BEG to verify a property of a small publish-subscribe program implemented using Java's `Observer` and `Observable` classes from the `java.util` library. Figure 3.1 shows classes `Subject` and `Watcher`, which play a role of observable and observer. The `Subject` class declares the field `obs` of type `Buffer`, shown in Figure 3.2, which is a container for `Watcher`s that are registered for the `Subject`. The `Watcher` class contains a bookkeeping field `registered`, which keeps track of whether the `Watcher` is registered on the `Subject`. Suppose, we are interested in reasoning about whether "Only registered `Watchers` are notified of `Subject` updates". This property is specified using Bandera Specification Language (BSL) in [15]. We can also assert that the `registered` field of `Watchers` is `true` at the point where a `Subject` calls `update()` (line 41 in Figure 3.1).

3.1.1 Unit and Property Specification

The user designates the unit under analysis by naming the collection of Java classes whose properties need to be verified. In general, selection of the classes in the unit is driven by the

```
1  public class Subject extends Observable {
2    protected boolean changed = false;
3    protected Buffer obs = new Buffer();
4
5    public void changeState(){
6      setChanged();
7      notifyObservers();
8    }
9    public synchronized void add(Watcher o){
10     obs.register(o);
11   }
12   public synchronized void delete(Watcher o){
13     obs.unregister(o);
14   }
15   public void notifyObservers(Object arg) {
16     Watcher cw;
17     Buffer lb = new Buffer();
18     synchronized (this) {
19       if (!changed)
20         return;
21       obs.copy(lb);
22       changed = false;
23     }
24     if (obs.size() != lb.size())
25       cw = null;
26     while (!lb.isEmpty()) {
27       cw = lb.removeFirst();
28       cw.update(this, arg);
29     }
30   }
31   protected synchronized void setChanged(){
32     changed = true;
33   }
34   public synchronized boolean hasChanged() {
35         return changed;
36   }
37 }
38 public class  Watcher implements Observer{
39   public boolean registered = false;
40   public void update(Observable o, Object arg) {
41     assert registered;
42   }
43 }
```

Figure 3.1: *Customized Observer-Observable Implementation*

```
1  public class Buffer extends Vector {
2    public void register(Watcher w) {
3      if (!contains(w))
4        super.addElement(w);
5      w.registered = true;
6    }
7    public void unregister(Watcher w) {
8      super.removeElement(w));
9      w.registered = false;
10   }
11   public Watcher removeFirst() {
12     Watcher result = elementAtIndex(0);
13     unregister(result);
14     return result;
15   }
16 }
```

Figure 3.2: *Customized Buffer Implementation*

properties that one wants to reason about. For our example and the mentioned property, `Subject` and `Watcher` should be in the unit.

3.1.2 Interface Discovery

The interface discovery step finds the two parts of the unit-environment interface: the unit interface and the environment interface.

To find the unit interface, the unit classes are analyzed to discover methods and fields of the unit that may be referenced by the environment. For our example, public methods `changeState()`, `add()`, `delete()`, `notifyObservers()`, `setChanged()`, and `hasChanged()` of class `Subject` and `update()` of the `Watcher` class may be invoked by the environment. In addition, `Watcher`'s field `registered` can be directly assigned to some values by the environment. The unit's exposed methods and fields are used by BEG to construct environment actions, which can be used to describe patterns of actions the environment may perform on the unit.

To discover the environment interface, the unit classes are analyzed to discover any external classes, methods, and fields referenced inside the unit. The external references drive the generation of the environment components that are directly referenced by the

unit. In our example, `java.util.Observable`, `java.util.Observer`, and `Buffer` are in the environment. Note that the actual environment may consist of more classes due to transitive class and method dependencies, for example, the `java.util.Vector` class is also in the environment as a supertype of the `Buffer`. To make environments more compact, BEG can be configured to generate environment components that are immediately referenced in the unit. BEG approximates the rest of the environment by incorporating summaries of classes that have an indirect effect on the unit into the generated environment components.

3.1.3 Driver Generation

BEG can be configured to generate universal drivers, which can perform all possible sequences of actions exposed by the unit interface. We show the code of the universal driver for the observer-observable example, automatically generated by BEG, in Appendix A.1.1; BEG is configured to instantiate two instances of each unit type and to create two threads performing all possible sequences of actions on the created instances. It is clear that universal drivers are not practical as they contain many nondeterministic choices and may contribute to the state space explosion.

Environment behavior can be constrained by environment assumptions. These assumptions can be specified or automatically extracted from the environment code, if available. In our case studies, the code for drivers is not initially available, therefore, we use specifications to constrain driver behavior. Given assumptions about sequences of environment actions, BEG generates a set of driver threads that implement the most liberal model that is consistent with the given assumptions (i.e., any computation in the model satisfies the assumptions and any computation that satisfies the assumption is a computation in the model). Figure 3.3 (top) illustrates an assumption with one instance of `Subject` and two `Watcher`s and a pair of threads whose behavior is given by regular expressions over method names with parameter values elided. The elided parameters can be interpreted as values that are selected nondeterministically from the possible values of the parameter type. The

```
environment {
  setup { 1 Subject; 2 Watcher; }
  driver-assumptions {
      re{
        Change: (changeState())*
        Register: (add() | delete())*
      }
  }
}
```

```
1  public class EnvDriver {
2    public static void main(String[] p0){
3      Subject s0 = new Subject();
4      Watcher w0 = new Watcher();
5      Watcher w1 = new Watcher();
6      new Change(s0, w0, w1).start();
7      new Register(s0, w0, w1).start();
8    }
9  }
10 public class Change extends java.lang.Thread {
11   public Subject s0;
12   public Watcher w0, w1;
13   public T0(Subject p0, Watcher p1, Watcher p2){
14     s0 = p0; w0 = p1; w1 = p2;}
15     public void run(){
16     while(Verify.randomBool())
17       s0.changeState();}
18 }
19 public class Register extends java.lang.Thread { ...
20   public void run(){
21     while(Verify.randomBool())
22       switch(Verify.randomInt(1)){
23       case 0:
24         s0.delete(Verify.randomObject("Watcher"));
25         break;
26       case 1:
27         s0.add(Verify.randomObject("Watcher"));
28         break;
29     }
30   }
31 }
```

Figure 3.3: *Observer-Observable User Assumptions and Driver Model*

first thread repeatedly calls the `changeState()` method on the only instance of `Subject` and the second thread calls any sequence of `add()` or `delete()` calls on the `Subject` object with a nondeterministically selected `Watcher`.

Figure 3.3 (bottom) also shows the generated drivers that capture the assumed behavior. The `EnvDriver` class allocates the specified instances and starts the execution of the two threads. Thread implementations model the assumption specifications by invoking *modeling primitives* that are interpreted by the underlying model checker as a nondeterministic choice over a set of values, as described in section 2.2.5.

3.1.4 Stub Generation

A user may specify assumptions for stubs in a way that is similar to driver specifications: the program actions, their sequence patterns, and data values are specified in a similar way. We show an example of user specifications for the `Buffer` class in Appendix A.1.2. However, if the implementation of the environment components is available, then static analysis techniques can be used to generate stubs automatically.

One may start with generation of empty stubs using the simplest settings in BEG. Appendix A.1.3 shows BEG's configuration and the `Buffer` model with empty stubs. This setting is useful for generation of library classes that do not have many side-effects with respect to the unit under analysis. In our particular example, the field `registered` is modified by the environment.

BEG's points-to and side-effects analyses can be applied to determine how the environment methods can influence the unit data [85, 87]. In our example, the analysis of the `Buffer` implementation calculates the following effects: methods `register` and `unregister` may/must side-effect the field `registered` of the `Watcher`. As described in section 2.1.3, *may* side-effects analysis calculates possible side-effects, whereas *must* side-effects analysis calculate side-effects that happen on all method executions.

Models are generated to reflect possible side-effects as calculated by the analyses. Fig-

```
1  public class Buffer{
2    public static Buffer TOP_OBJ =
3      new Buffer();
4    public void register(Watcher p){
5      if(Verify.randomBool()
6        p.registered = true;
7    }
8    public void unregister(Watcher p){
9      if(Verify.randomBool())
10       p.registered = false;
11   }
12   public Watcher removeFirst(){
13     return Verify.
14            randomObject("Watcher");
15   }
16 }
```

```
1  public class Buffer{
2    public static Buffer TOP_OBJ =
3      new Buffer();
4    public void register(Watcher p){
5      p.registered = true;
6    }
7    public void unregister(Watcher p){
8      p.registered = false;
9    }
10   public Watcher removeFirst(){
11    return Verify.
12           randomObject("Watcher");
13   }
14 }
```

Figure 3.4: *Buffer's Models Based on May and Must Side-Effects Analysis*

ure 3.4 (left) shows the Buffer model generated using BEG's *may* side-effects analysis, which calculates that on some paths of the method `register()`, it assigns its parameter's `registered` field to `true` and, similarly, on some paths of the method `unregister()`, it assigns its parameter's `registered` field to `false`. The possibility of a side-effect, produced by the *may* side-effects analysis is encoded in Java by enclosing the side-effecting statement with `if(Verify.randomBool()` (lines 3,4 and 7,8). The *must* side-effects analysis option in BEG refines results of the may side-effects analysis by factoring out side-effects that occur on all executions; such side-effects are encoded in Java as unconditional assignments, thus producing more precise stubs, as shown by lines 5 and 8 in Figure 3.4 (right).

If the analysis cannot calculate the concrete values on the right hand side of side-effecting assignments or return statements, then approximate values are used. For example, the return value of `removeFirst()` is approximated using `randomObject("Watcher")`, which is interpreted as a nondeterministic choice over the heap instances of `Watcher` type.

To minimize the state space of the environment, for each environment type BEG generates `TOP_OBJ` field. This field denotes a single instance of the environment class and is used during code generation for the environment, i.e., any time the environment needs a

```
1  public class Buffer{
2    Object elementData[];
3    public void register(Watcher p){
4      //must side-effects
5      elementData[TOP_INT] = p;
6      p.registered = true;
7      //may side-effects
8      ...
9    }
10   public Watcher removeFirst(){
11     //may side-effects
12     elementData[TOP_INT] = null;
13     ...
14     //return locations
15     return Verify.
16     randomReachable("Watcher",this);
17   }
18 }
```

```
1  public class Buffer{
2    Object elementData[];
3    int count;
4    ...
5    public void register(Watcher p){
6      elementData[count] = p;
7      count++;
8      p.registered = true;
9    }
10
11   public Watcher removeFirst(){
12    return Verify.
13    randomReachable("Watcher",this);
14   }
15 }
```

Figure 3.5: *Buffer's Containment Models: Automated and Refined*

value of type $C \in E$, it uses the `C.TOP_OBJ` field. This mechanism avoids creating multiple environment objects by the environment itself. It also allows one to mark the environment objects for further processing by abstraction and symbolic execution techniques. Note that the unit can create multiple instances of environment classes by calling regular constructors of the environment classes. Section 7.3.2 shows more examples of using `TOP_OBJ` fields in the environment models.

3.1.5 Model Checking and Refinement

After the environment models are generated, we combine them with the code of the unit and use JPF or Bogor to verify the unit properties. Model checking the observer example from Figure 3.1 with JPF using the environment models from Figures 3.3 and 3.4 (on the right), and the mentioned property yields a spurious counter-example where an unregistered `Watcher` is notified of an update. This is due to the imprecision of the generated code for the `removeFirst()` method, which can return any allocated instance of type `Watcher`.

To produce a more precise model of the `Buffer` class, we use BEG's containment analysis, designed to track containment properties of environment components. A compo-

nent with the containment properties can store objects of the unit type, thus partitioning the heap into objects that are *reachable* from the container and objects that are not reachable from it. The left side of Figure 3.5 shows the model generated by BEG using the containment analysis tuned to track side-effects to array-type fields. Note that BEG emits `elementData[TOP_INT] = p` to denote the addition of the method's parameter to the `elementData` array, inherited by the `Buffer` from the `Vector`. BEG also emits `randomReachable("Watcher", this)` as a return value of the method `removeFirst()`. One can go over the code generated by BEG and refine it, e.g., `elementData[TOP_INT]` can be refined as `elementData[count]`. The right side of Figure 3.5 shows the manually refined Buffer stub.

Boosting the precision of the `Buffer` model, by using the container model of the `Buffer`, eliminates the spurious counterexample and reveals a property violation due to a race condition in the implementation of `notify()` that is due to the intentional limitation of the scope of the `synchronized` statement for improved performance.

Section 7.2.2 contains more examples showing how containment can be used to boost the precision of generated stubs for the Swing and AWT libraries. The user can specify the fields to track, e.g., `Componenet[] component` field of the `java.awt.Container` class. For APIs known a priori, e.g., for container classes from the `java.util` package, BEG can be configured to treat such classes as part of the unit and preserve calls from the environment classes to containers such as `Lists`, `HashMaps`, etc. This approach is used to develop stubs for the `javax.servlet.http` APIs, described in section 4.2.3

3.2 Environment Generation Methodology

When model checking the observer-observable example, we followed a methodology that can be described by the following steps:

1. Module isolation and property specification

2. Environment generation

   (a) Interface discovery

   (b) Driver generation

   (c) Stub generation

3. Model Checking and Refinement

   - Error detected
     - Spurious error: go to 2
     - Actual error: fix the error, go to 3
   - Incomplete or Verified: check coverage
     - Desired coverage: done
     - Poor coverage: go to 2

This thesis addresses automated support for step 2, environment generation. We use existing Java model checking frameworks to perform model checking in step 3. We give useful insights on how to approach unit selection and model refinement, however, automated support for these steps is beyond the scope of this thesis. We discuss ideas for automation of unit selection and model refinement in section 8.2, Future Work.

3.2.1 Unit and Property Specification

The unit selection is driven by the unit properties: classes mentioned in the properties should be included in the unit. Domain-specific knowledge can also be used to specify the unit. For example, for GUI and J2EE applications, one can specify the unit as the application-specific code, treating GUI and J2EE libraries as the environment that needs to be stubbed out. In Chapter 4, we describe our domain-specific methodology for GUI and J2EE applications.

3.2.2 Interface Discovery

This step calculates the syntactic unit-environment interface, which consists of the unit interface and the environment interface.

In general, the unit interface consists of all public methods and fields. However, for specific domains, the unit's interface can be restricted to domain-specific APIs. For instance, for GUI and J2EE applications, driver actions can be defined as special *event-handling* methods that process user inputs such as button clicks. We discuss domain-specific APIs in Chapter 4. The set of methods and fields in the unit interface is used to generate a universal driver or validate driver specifications.

The environment interface consists of *all* external classes, methods, and fields referenced by the unit. The syntactic environment interface drives stub generation for the components that are directly referenced in the unit, potentially omitting generation of many classes that are indirectly used by the unit. The environment interface is also used to validate user specifications for stubs.

3.2.3 Driver Generation

BEG has support for generating universal drivers and drivers generated from user specifications. Section 5.3 describes BEG's support for user specifications in detail. In short, BEG supports specification of different aspects of driver behavior: synchronization (e.g., number of driver threads), control (e.g., sequences of actions performed on the unit), and data (e.g., driver and stub inputs).

One can start with universal drivers or drivers generated from user specifications and refine them. In this work, we start with drivers generated from user specifications, as universal drivers typically do not allow for tractable model checking.

3.2.4 Stub Generation

BEG has support for generating the following types of stubs: empty, universal, based on user specifications, and based on static analysis results. Currently, BEG performs side-effects analysis, which can be constrained based on domain-specific information. Chapter 4 describes domain-specific information for GUI and J2EE components, and Chapter 5.4 describes BEG's side-effects analysis, including its customization for GUI and J2EE libraries.

In this work, we start with empty stubs and keep enhancing them with additional behavior such as data and control effects. As the last step, slicing can be used to produce stubs with all possible unit-environment dependencies. Here is the hierarchy of stubs used in this thesis:

1. Empty stubs: no effects
2. Data effects: domain-specific side-effects
3. Control effects: callbacks
4. Slicing

Our experience shows that step 2 is very effective, especially if tuned for specific domains. For most case studies presented in Chapter 7, step 2 was sufficient to uncover errors or produce the desired coverage results.

3.2.5 Model Checking and Refinement

The generated environment may exercise too many unit behaviors to allow for tractable model checking or too few to cover all interesting behaviors. In both cases, the environment may need refinement: if the environment is too large, it may need to be reduced; if an error is known to exist but the environment is too restrictive and masks the error, then the environment needs to be expanded. Going in either direction may not be trivial, however, in some specific instances, it is possible to design a systematic approach to reduce or expand the environment based on domain-specific knowledge and model checking results.

- Unit Refinement: If too many dependencies exist between the unit and its environment classes, it is useful to redefine the module by moving tightly coupled classes from the environment to the unit. In this work, we used this approach when static analysis calculated too many side-effects for some environment classes, as described in section 7.3.

- Driver Refinement: In this work, we started with drivers synthesized from user specifications. In some instances, we needed to refine the drivers based on branch coverage results, as described in 7.3. For example, if a unit method is not getting covered, it can be added to the alphabet of driver actions.

- Stub Refinement: We found that for stubs, it is convenient to start with empty stubs and keep enhancing them with specific features. An alternative would be to start with universal stubs or stubs calculated by slicing. Our experience shows that universal stubs are largely impractical, whereas stubs based on slicing may retain too many dependencies between the unit and its environment. Our case studies show that for well defined units, calculating data effects of the environment is usually sufficient to detect errors. Also, for specific domains, e.g, for GUI and web applications, we built a specific hierarchy of features to be added to stubs; we describe these in Chapter 4.

To summarize, the following is our environment generation methodology:

- Generic Domain
  - Interface: All public methods and fields
  - Drivers: Constrained by user specifications
  - Stubs: Effects to data of unit type
- Specific Domain
  - Interface: Domain-specific APIs

- Drivers: Constrained by user specifications
- Stubs: Effects to domain-specific data

Overall, both driver and stub refinement can be customized for specific domains. One needs to study the domain and mine for domain-specific features that are important to preserve while modeling the domain. These features can be used to build the hierarchy of refinement steps, which can be automated. In the next Chapter, we describe two specific domains: Swing/AWT and J2EE frameworks. We describe domain-specific APIs that can constrain the alphabet of driver actions and domain-specific data to constrain BEG's side-effects analysis.

Chapter 4

Domain-Specific Environment Generation

An increasingly important class of object-oriented software systems are *frameworks*. Frameworks provide for large-scale reuse of functionality by collecting threads of control, operations and data structures that relate to a specific problem domain (e.g., Swing and AWT are Java frameworks that supports the development of Graphical User Interfaces (GUI), and J2EE is a Java framework for developing web applications). Frameworks present rich interfaces that allow application-specific processing to be coordinated through the framework. Frameworks are difficult to test due to the complexity of their interfaces and the degree of parameterization that is possible to configure their behavior. Current state-of-the-practice in framework testing relies on the use of groups of use cases to drive test case generation. In such a setting, BEG can be a valuable tool because it enables synthesis of drivers that capture multiple framework use cases. Furthermore, the use of nondeterminism in assumption specifications allows drivers to span multiple framework configuration settings. This offers the advantage of allowing configuration-independent properties to be analyzed without having to enumerate combinations of configuration settings.

In this chapter, we study two popular frameworks: Swing/AWT and J2EE. We present domain-specific knowledge and how it was used to customize environment generation for these domains.

4.1 Environment Generation for GUI Applications

GUIs are often employed to guide users through the interactions with an underlying application. GUIs can be effectively used in Human-Computer Interaction (HCI) systems to ease a user's task by presenting the behavior of the underlying application at a high level of abstraction, hiding potentially overwhelming details from the user, and by enabling/disabling GUI components to guide the user through the task.

Traditional approaches to validation of GUI aspects in HCI systems involve prototyping and live-subject testing. These approaches are limited in their ability to cover the set of possible human-computer interactions that a system may allow, since patterns of interaction may be long running and have large numbers of alternatives. In this section, we present environment generation customized for model checking GUI applications, where the behavior relevant to *interaction orderings*, enforced by a GUI, is preserved. This section and the case studies in section 7.2 are based on [26].

Verification of HCI systems is a significant challenge. While simplifying the interaction between a user and the underlying application, GUIs contribute to the complexity of the entire system by engaging in the interactions with the user and the underlying application. GUIs are usually written using frameworks such as the Swing and AWT toolkits, which simplify the programming effort but complicate verification of the end product. As a result, model checking of GUI applications may be intractable due to the complexity of the toolkit components used for the GUI implementation, infinite data domains in the underlying application, and complex semantics of the interaction between multiple systems: the user, the GUI, the underlying application, and the task the user is trying to achieve [18].

Despite the complexity of HCI applications, we exploit domain-specific knowledge to identify appropriate driver and stub generation techniques for GUI applications. Next, we discuss typical GUI frameworks features that influenced our environment generation approach. While there are many frameworks used for GUI development, in this section we concentrate on programs with GUIs implemented in the Java Swing and AWT frameworks.

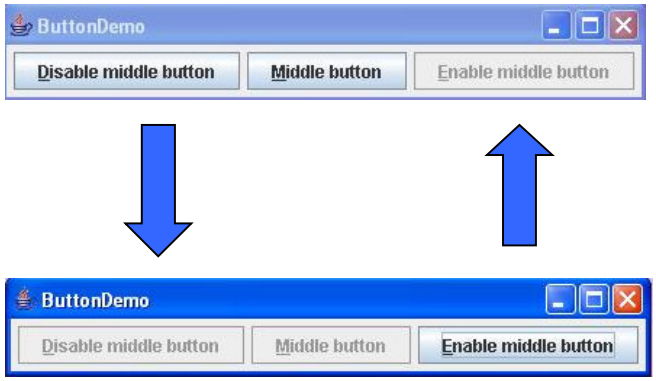

Figure 4.1: *ButtonDemo GUI States*

We believe that many of the high-level ideas presented in this section apply to other GUI frameworks.

4.1.1 Example: Button Demo

Swing and AWT are object-oriented frameworks, where windows, widgets on a window (e.g., buttons, selections, text entry boxes), the text and color associated with widgets and windows, and numerous additional attributes are all defined by instantiating framework classes. A typical GUI application creates a number of windows and widgets. As the user interacts with the application, the number of windows and widgets available for the interaction changes. A user can select any input action that is enabled in a given state of the GUI.

Figure 4.1 shows the GUI of a simple button demo example, taken from the SUN's Swing tutorial [83]. This simple GUI has three clickable buttons and only two states. In the initial state, the left and middle buttons are enabled and pressing on the left button disables the middle button and enables the right button. After pressing on the enabled right button, the GUI goes into its initial state. Pressing on the enabled middle button does not change the

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  public class ButtonDemo extends JPanel implements ActionListener {
5    protected JButton b1, b2, b3;
6    public ButtonDemo() {
7      b1 = new JButton("Disable␣middle␣button", null);
8      b1.setVerticalTextPosition(AbstractButton.CENTER);
9      b1.setHorizontalTextPosition(AbstractButton.LEFT);
10     b1.setActionCommand("disable");
11
12     b2 = new JButton("Middle␣button", null);
13     b2.setVerticalTextPosition(AbstractButton.BOTTOM);
14     b2.setHorizontalTextPosition(AbstractButton.CENTER);
15
16     b3 = new JButton("Enable␣middle␣button", null);
17     b3.setActionCommand("enable");
18     b3.setEnabled(false);
19
20     //Listen for actions on buttons 1 and 3.
21     b1.addActionListener(this);
22     b3.addActionListener(this);
23
24     b1.setToolTipText("Click␣this␣button␣to␣disable␣the␣middle␣button.");
25     b2.setToolTipText("This␣middle␣button␣does␣nothing␣when␣you␣click␣it.");
26     b3.setToolTipText("Click␣this␣button␣to␣enable␣the␣middle␣button.");
27
28     add(b1); //Add Components to this container
29     add(b2); //using the default FlowLayout
30     add(b3);
31   }
32   public void actionPerformed(ActionEvent e) {
33     if (e.getActionCommand().equals("disable")) {
34       b2.setEnabled(false);
35       b1.setEnabled(false);
36       b3.setEnabled(true);
37     } else {
38       b2.setEnabled(true);
39       b1.setEnabled(true);
40       b3.setEnabled(false);
41     }
42   }
43   public static void main(String[] args) {
44     JFrame frame = new JFrame("ButtonDemo");
45     ...
46     frame.getContentPane().add(new ButtonDemo(), BorderLayout.CENTER);
47     frame.pack();
48     frame.setVisible(true);
49   }
50 }
```

Figure 4.2: *Button Demo Example (excerpts)*

state of the GUI. This example has no underlying application, yet, it demonstrates many important features of GUI applications.

Figure 4.2 shows the implementation of the `ButtonDemo` class. This example creates one `JFrame` (line 44), one `JPanel` (line 46), and three buttons of type `JButton` (lines 7, 12, and 16).

4.1.2 Domain-Specific Knowledge

Swing and AWT Components

The essential step in modeling GUI components is identifying their *abstract state*. The abstract state of each GUI component is defined by the values of the fields we model; the abstract state of the entire system is defined by the state of each constituent component. For example, an instance of `java.awt.Component` defines values for over eighty fields of various types (e.g., `java.awt.Color background`, `int width`). Capturing an actual state of the `Component` object by specifying values of all of its fields is very expensive. Fortunately, for verification of interaction ordering behavior, we only need to record the values of fields that relate to the logical state of the component and not its *look and feel*. Next, we describe the properties of Swing and AWT components that are relevant to interaction ordering.

Containment Hierarchy A typical Java GUI consists of a number of components (frames, panels, buttons, labels). There is usually a *top-level container* that provides a place for the other components to paint themselves. Some of the most commonly used top-level containers are `JFrame` (e.g., in the `ButtonDemo` example) and `JApplet` (e.g., in the Autopilot example described in section 7.1). There are also *intermediate containers* such as `JPanel` that simplify the positioning of *atomic* components such as buttons and labels. The atomic components are usually the components whose role is to get inputs from the user.

Atomic containers are usually added to the intermediate containers, which in turn are added to the top-level containers using special `add()` methods (e.g., lines 28-30 add buttons to the `ButtonDemo` and line 46 adds the `ButtonDemo` to the top-level `JFrame`). Events are

usually propagated through the containment hierarchy. Therefore, when modeling GUI components, we need to preserve their *containment* properties.

Modality A *modal* dialog is one that restricts the next user interaction to the enabled actions on that dialog; all other actions are disabled. The `ButtonDemo` example does not create modal dialogs, but they are commonly used to display error messages or prompt a user for additional information. Until the user dismisses the modal dialog, he is restricted to using that window only.

Modality is tracked by the boolean field `modal` of the `java.awt.Dialog` class. The `modal` field can be set using the method `setModal()` or when invoking one of the `Dialog`'s constructors that take the boolean `modal` as one of the arguments.

Enabledness and Visibility Once a user chooses a window to work with, he is then able to select from that window's components that are both *visible* and *enabled*. Disabled or invisible components do not allow for user selection. Visibility and enabledness is set using methods `setVisible()` (e.g., line 48) and `setEnabled()` (e.g., lines 34-36 and 38-40), which modify the boolean fields `enabled` and `visible` of the `java.awt.Component` class.

Selection In addition to buttons, there are tabs and radio buttons, which show selection: usually, only one item at a time may be selected. Selecting a new component deselects the previously selected one. This is called a *single selection model*. Selection is implemented by consulting the containment and selection information when updating the component states for the radio buttons. More specifically, it is implemented using the boolean field `selected` and can be set using the method `setSelected(boolean b)` of the GUI components (e.g., `javax.swing.JRadioButton`). In addition, radio buttons are added to a button group (e.g., `ButtonGroup`), which keeps track of a currently selected button (e.g., `AbstractButton` field `selection` of the `ButtonGroup` class).

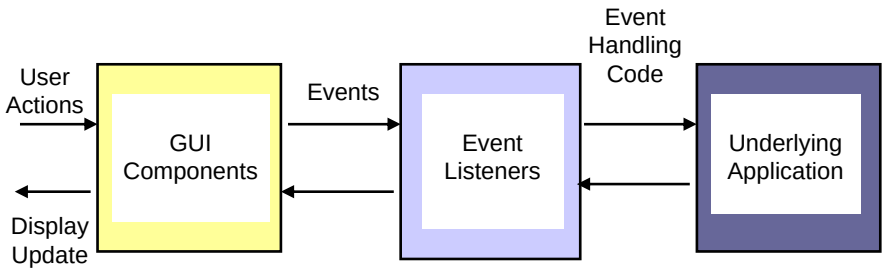

Figure 4.3: *Swing/AWT Event Handling Mechanism*

Look and Feel Swing types also provide an enormous variety of options for controlling the visual aspects of the GUI. For example, in Figure 4.2, lines 8, 9, 13, and 14 set the vertical and horizontal positions of the left and middle buttons. These features are a part of the look and feel properties and are not relevant for reasoning about the interaction orderings.

Event-Handling

GUI applications are reactive systems that take sequences of user actions (e.g., mouse clicks, mouse movements) and produce changes in the state of the GUI and/or the underlying application. On the implementation level, each time a user clicks on a button or performs any other action on a screen of a GUI, an *event*, an object of type `Event`, is fired and processed by the *event-handling methods* of the *event-handlers*, objects of type `EventListener`. Any GUI component can be notified of the event if it implements the `EventListener` interface and is registered as an event listener on the appropriate event source. The event-handling methods examine the event object and, depending on the information it carries, change the state of the GUI and/or the underlying application. The event objects carry various information, e.g., the type of the button clicked, the command associated with that specific button, the text a user entered into the text field. Figure 4.3 shows a high-level view of the event-handling

mechanism in Swing and AWT. The framework executes a cyclic *event-dispatching thread*, which processes each event in turn and invokes the associated event-handling methods. The event listening mechanism used in Swing/AWT is an example of the observer-observable pattern, similar to the publish-subscribe example described in section 3.1.

In Figure 4.2, the `ButtonDemo` class implements `ActionListener` by providing implementation of `actionPerformed(ActionEvent e)` (lines 32-42). Lines 21-22 implement registration of the listener to buttons `b1` and `b3`. Clicking on `b1` or `b3` creates an object of type `ActionEvent` and invokes the `actionPerformed()` method of the `ButtonDemo`, passing the event as an argument to the method.

The GUI events can be divided into two categories: *logical* events correspond to user actions that require interaction with the underlying application or GUI control-logic, whereas *low-level* events indicate actions that are primarily handled automatically by the default GUI framework, e.g., listeners that highlight a component or display a tooltip when a mouse is moved over it. We focus on the logical events and their handlers in this work, although the low-level events can be treated using the same mechanisms.

4.1.3 Domain-Specific Methodology

Our environment generation approach consists of partitioning an HCI application into three parts: the Swing/AWT framework, the *GUI implementation* (i.e., the application-specific GUI setup and event-handler code that interacts with Swing/AWT to configure the structure of the GUI and to implement its control-logic), and the *underlying application* (i.e., the code that is common to GUI and command-line versions of an application). By decomposing an application in this way we can target each part with a different technology for extracting a faithful and appropriately precise model of its behavior. In this setting, the GUI implementation is the unit under analysis, the missing user component is the driver, and the Swing/AWT framework components are stubbed out. The underlying application can be either included into the unit under analysis or stubbed out, depending on the types of

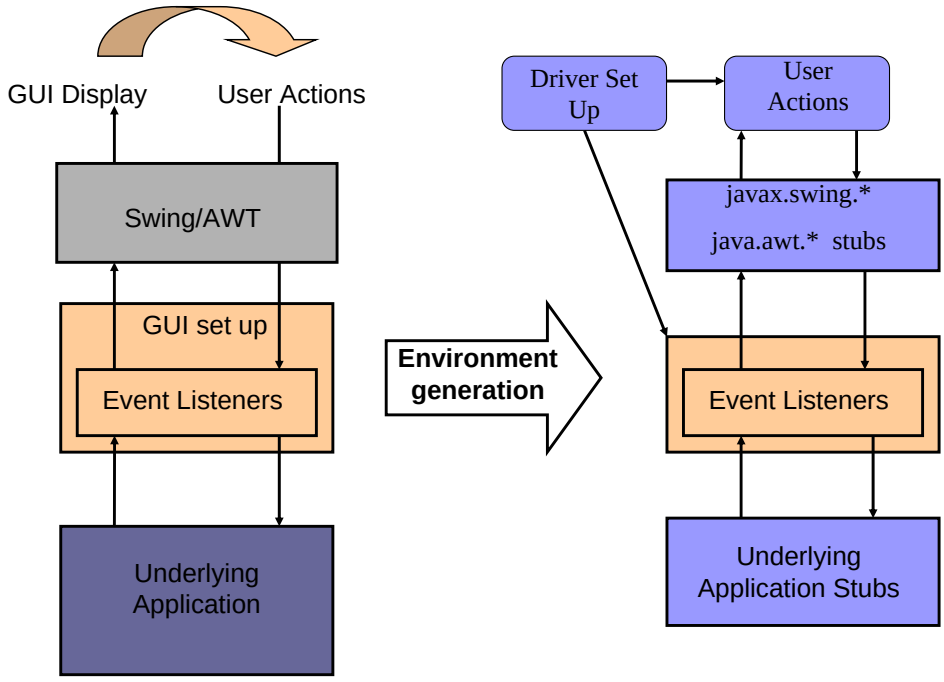

Figure 4.4: *Environment Generation for GUI Applications*

properties being checked. In section 7.1, we present the autopilot tutor example, for which we preserve the state of the underlying autopilot and in section 7.2, we discuss case studies with an underlying application stubbed out.

The left-side of Figure 4.4 illustrates the three-layer structure of a Swing/AWT application. The framework owns the main execution thread, controls the rendering of the display images, and processes user inputs to produce events that are relevant to the application. The GUI implementation is comprised of all of the application code that directly manipulates Swing types, for example, to configure the structure of the GUI and to define and register event-handler methods. The underlying application is the remainder of the application that is, by definition, not directly dependent on Swing types.

Our approach is to develop a single model of the Swing/AWT framework's interaction ordering-related behavior. Intuitively, the different layers of a Swing/AWT application exert different degrees of influence on the behavior of the overall program relative to interaction orderings. Towards this end, we extend BEG to preserve information about Swing/AWT interaction ordering-related types and fields. Next, we discuss our environment generation methodology customized for the Swing/AWT frameworks.

Interface Discovery

The task of the interface discovery step is to find all methods and fields that can be exercised by the environment, in this case a GUI user, and to find all external references, including Swing/AWT libraries. In terms of finding external references, BEG did not need any tuning, but for discovering methods that can simulate user actions, some tuning was needed. For this domain, the unit interface consists of special event-handling methods, e.g., `actionPerformed()`, of the event handling classes.

Driver Generation

BEG can generate two types of drivers: universal and synthesized from user specifications. The use of specifications did not need any tuning for this domain. The universal driver generation, however, needed special handling.

Next, we give an example of a universal driver for the `ButtonDemo` class. The entry points to the unit under analysis are the special event-handling methods, e.g., `actionPerformed()` for the `ButtonDemo` example. Using these methods, one can write specifications to describe sequences of user events, e.g., `(actionPerformed())*`, which describes a universal driver for the `ButtonDemo`.

Figure 4.5 shows the driver code for the `ButtonDemo` example, with the set up section taken directly from the `main` method of the `ButtonDemo` class, with addition of creation of two events. The assumptions section reflects the above specification. We added an assertion to check for the property "buttons b1 and b3 are never enabled at the same time".

```
1  import env.javax.swing.*;
2  import env.java.awt.*;
3  import env.java.awt.event.*;
4  import gov.nasa.jpf.jvm.Verify;
5  public class EnvDriver{
6    public static void main(String[] args){
7      //set up, taken directly from GUI implementation
8      JFrame frame = new JFrame("ButtonDemo");
9      ...
10     ButtonDemo buttonDemo = new ButtonDemo();
11     frame.getContentPane().add(buttonDemo, BorderLayout.CENTER);
12     frame.pack();
13     frame.setVisible(true);
14
15     ActionEvent actionEvent1 = new ActionEvent( "disable");
16     ActionEvent actionEvent3 = new ActionEvent("enable");
17
18     //assumptions
19     while(Verify.randomBool()){
20       assert(!(buttonDemo.b1.isEnabled() && buttonDemo.b3.isEnabled()));
21       ActionEvent event = (ActionEvent)Verify.randomObject(
22                           "env.java.awt.event.ActionEvent");
23       buttonDemo.actionPerformed(event);
24     }
25   }
26 }
```

Figure 4.5: *ButtonDemo Universal Driver (excerpts)*

The `ButtonDemo` class is a simple GUI example, with one event-handler, which is registered on one component, which in turn is always visible and enabled. In general, we need to take into account the information about the GUI components' visibility and enabledness, i.e., the constraints the GUI puts on the user's actions. Since Swing/AWT event-handling APIs are known a priori, we can write a driver that will work for any GUI application, written using Swing/AWT. We present such a driver in section 7.2 and use it to model check a collection of small GUI examples from [83]. The universal driver presented in section 7.2 takes into account GUI components' visibility and enabledness in each state. In section 7.1, we present a case study where regular expressions are used to constrain user actions according to given use case scenarios.

Stub Generation

Containment To preserve the containment information of the GUI components, we configure BEG to perform containment analysis, which can keep track of side-effects to array-type fields or callbacks to the fields of container type from the `java.util` package. This is similar to the container model used for the `Buffer` class, described in section 3.1; more examples are given in section 7.2.2.

Modality Modality is modeled by preserving the `modal` field of `java.awt.Dialog` and modeling side-effects of `setModal()` and other methods on this specific field. We customize BEG's side-effects analysis to keep track of side-effects to the `modal` field. Therefore, we keep track of modality as part of the abstract state of dialogs. In addition, we keep track of all available windows in the system in two data structures: a set of windows that do not restrict user interactions (i.e., frames and non-modal dialogs) and a stack for restrictive windows (i.e., modal dialogs). At each step, if the second structure is not empty, the modal dialog on the top of the stack represents the window a user may interact with. If there are no modal dialogs open, then the user may interact with any window from the first set.

Visibility and Enabledness We model these aspects of GUI components by including per widget visibility and enabledness booleans. We configure BEG to keep track of side-effects to `enabled` and `visible` boolean fields of Swing/AWT classes.

At each step, the user may choose among the visible and enabled children of a top-level window. While certain components may not be displayed in a given state (and hence reasonably considered *invisible*), we consider a logical notion of visibility defined as follows : a component is *visible* if either it is visible on the display in the current state or it can be made visible by selecting a series of visible components starting in the current state. Thus, we consider the set of all components that lie on a path which consists of visible components to be logically visible.

Selection We keep track of selection by modeling the `selected` field of radio buttons, their button group, and the selected button within a button group. We customize BEG to keep track of side-effects to these fields.

Event Handling The event-handling code is preserved as part of the unit under analysis. The library methods used to register event listeners on the events are modeled to preserve the containment properties, e.g., `addListener()` methods are implemented similar to `add` methods of containers. We configure BEG to keep track of containment properties with respect to the listener components.

Look and Feel Fields that implement the look and feel properties are not modeled; methods that have side-effects on such fields are modeled using empty stubs.

Putting all of these features together, we configure BEG to produce summaries of side-effects on the interaction ordering-related framework data for each method in the Swing/AWT APIs. The resulting combination of models safely captures the event-related GUI behavior while abstracting other aspects of the GUI, e.g., color, shape, size, and the underlying application. Furthermore, the model retains the structure of an event-driven Swing/AWT system. This safe, but approximate, model is used as a starting point for the manual development of a more precise model, if needed. This model is reused across multiple analyses, hence the cost of its construction can be amortized. Sections 7.1 and 7.2 present our experience with model checking applications written using Swing and AWT libraries.

4.2 Environment Generation for J2EE Applications

In this section, we describe environment generation for a specific class of web applications. While there are many frameworks used for development of web applications, we base our discussion on the J2EE framework [81], used to implement *enterprise* applications. Next, we describe features of J2EE applications relevant to environment generation and model

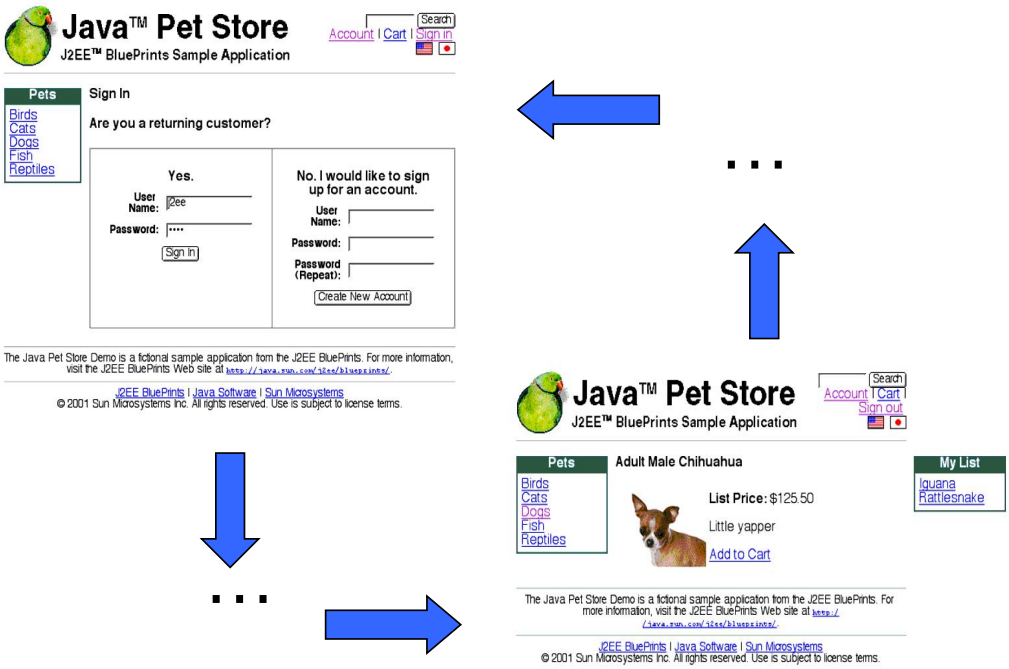

Figure 4.6: *Pet Store Sign in and Item Screens*

checking. We believe that many of the high-level ideas presented in this section apply to model checking web applications written using other frameworks, e.g., Struts [80] and Hibernate [40].

4.2.1 Example: SUN's Pet Store

As an example, we use SUN's Pet Store [82], an open-source application distributed with SUN's AppServer. The Pet Store example is a typical enterprise application, which allows users to create an account, sign-in to their account, browse through a catalog of pets, add chosen pets into a shopping cart, place their order, and sign out. The Pet Store has several screens, which constrain and guide the user through the sequences of actions, similar to a GUI in Swing/AWT applications. Figure 4.6 shows the Sign-in (top-left) and Item (bottom-right) screens of the Pet Store example. An example of a functional property we would like

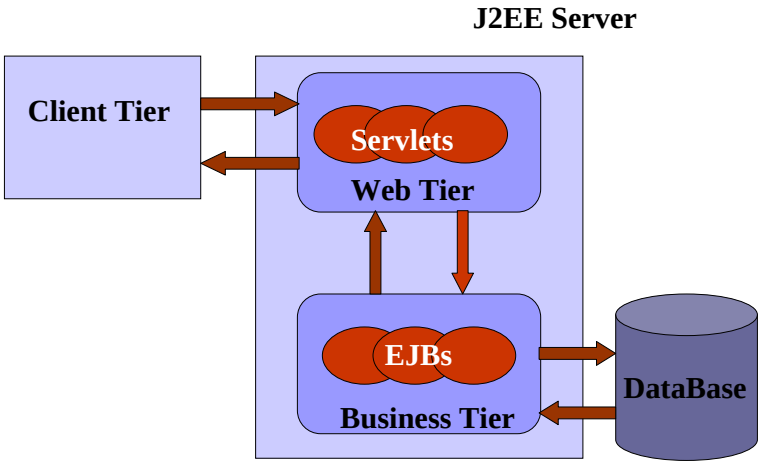

Figure 4.7: *J2EE Applications Architecture*

to check for the Pet Store example is "after check out, the cart becomes empty".

4.2.2 Domain-Specific Knowledge

Architecture

Enterprise applications are open distributed multilayered systems, usually comprised of artifacts written in several languages (e.g., Java, HTML, XML). A typical enterprise application is distributed over three different locations: a client machine, a J2EE server machine, and a database machine. Therefore, enterprise applications are considered to have three tiers. Figure 4.7 shows the architecture of a typical enterprise application. The three layers embody different functional aspects of the enterprise applications:

- The *client tier* is in charge of the interface with a user. It may include a web browser, web pages, or applets.

- The *server tier* processes data, which flow from a client to a database and vise versa. The server tier usually contains two layers: the *web tier*, which consists of Java Servlets or JavaServer Pages (JSPs), and the *business tier*, which handles business logic by

employing the Enterprise JavaBeans (EJBs).

- The *database* tier is in charge of data persistence. It is optionally deployed on a separate machine.

To apply Java model checking techniques, e.g., JPF, all non-Java components, including a user, need to be represented as pure Java implementations. In addition, the distributed nature of an application needs to be dealt with so that the resulting Java model is non-distributed but preserves the behavior of the original application relevant to its logical state.

Next, we describe J2EE components and the event-handling mechanism of enterprise applications in greater detail.

J2EE Components

Before the web and business tier components can be executed, they are deployed within the web and EJB *containers*. These containers serve as a run-time environment for servlets and EJBs. They are configured according to *descriptor files*, usually written in XML and packaged with an enterprise application.

Web tier The web tier contains servlets and JSPs, which filter and process user requests. The web container is configured according to descriptor files containing information pertaining to the web tier, e.g., security and authentication mechanisms.

Business tier The business tier contains EJBs, which process data flowing from a client and store it in a database. Likewise, EJBs retrieve data from the database, process it, and send it back to the client. The EJB container is configured according to descriptor files, containing information about the business tier, e.g., EJB categories and transaction management.

There are several categories of EJBs: *session* beans are in charge of information that pertains to a single client session, *entity* beans are in charge of data persistence, and *message-driven* beans allow for asynchronous message processing.

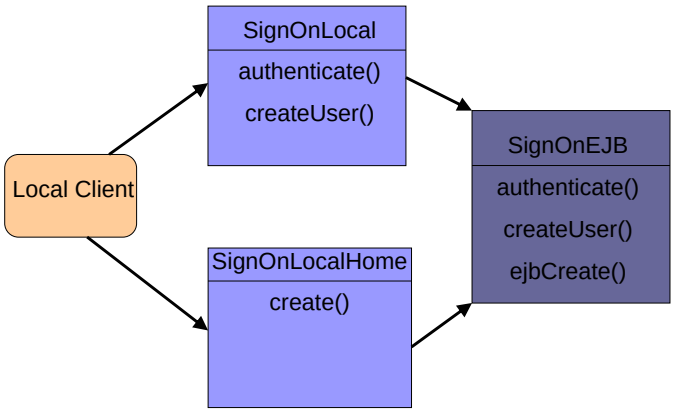

Figure 4.8: *Interfaces for SignOnEJB with local access*

Due to the distributed nature of enterprise applications, session and entity beans are designed to be accessed by a client using the bean's interfaces. There are two types of access: *remote* and *local*. A bean with a remote access has remote and home interfaces; a bean with a local access has local and local home interfaces. Remote and local interfaces define the bean's business methods; home and local home interfaces define the bean's life cycle and finder methods. The Pet Store EJBs support local access. Figure 4.8 shows an example of the local and local home interfaces for `SignOnEJB`, used when the `createUser` action, available on the signIn screen, is performed. We discuss this action in more detail later.

Database The database is an important part of J2EE applications. The database access APIs are implemented by the `java.sql` package. The database works as a container that holds application data.

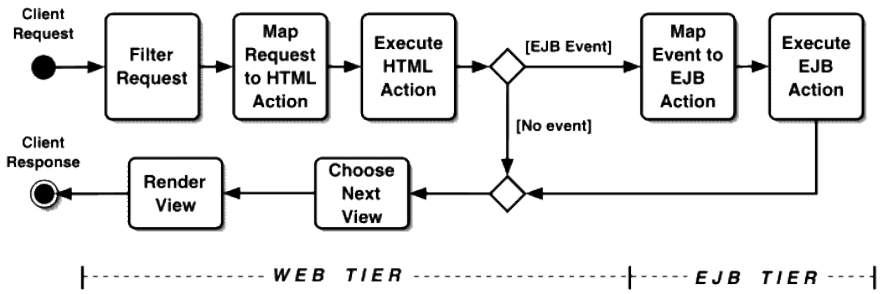

Figure 4.9: *Pet Store Event Handling*

```
<url-mapping url="createuser.do" screen="create-customer.screen" >
        <web-action-class>...web.actions.CreateUserHTMLAction</web-action-class>
</url-mapping>
...
<event-mapping>
        <event-class>...events.CreateUserEvent</event-class>
        <ejb-action-class>...ejb.actions.CreateUserEJBAction</ejb-action-class>
</event-mapping>
```

Figure 4.10: *Pet Store Descriptor File mappings.xml (excerpts)*

Containment Similar to the Swing/AWT framework, the J2EE components' features include containment properties. Many J2EE classes are implemented to hold information, e.g., in the package `javax.servlet.http`, `HttpServletRequest` objects contain user data flowing from a web browser to the web tier, `HttpServletResponse` objects contain the application data flowing from the web tier back to the user, `HttpSession` holds the session data, and `HttpSessionContext` holds the context information.

Event Handling

The event-handling mechanism of enterprise applications is similar to the event-handling mechanism of GUI applications: there are events and their corresponding event-handling classes. Unlike in GUI applications, there could be one or two levels of event handling: one through the web tier and one through the business tier. Figure 4.9 shows the two-tier event-handling mechanism for the Pet Store example. The client request flows to the web tier,

```
1  public class CreateUserHTMLAction extends HTMLActionSupport {
2    public Event perform(HttpServletRequest request)
3      throws HTMLActionException {
4
5      String userName = (String)request.getParameter("j_username");
6      String password = (String)request.getParameter("j_password");
7      String password2 = (String)request.getParameter("j_password_2");
8      ...
9          if (userName != null && password != null)
10             return new CreateUserEvent(userName,password);
11         return null;
12     }
13 }
14 public class CreateUserEJBAction extends EJBActionSupport {
15   public EventResponse perform(Event e) throws EventException {
16     CreateUserEvent cue = (CreateUserEvent)e;
17     String userName = cue.getUserName();
18     String password = cue.getPassword();
19     ...
20     ServiceLocator sl = new ServiceLocator();
21     SignOnLocalHome home =(SignOnLocalHome)sl.getLocalHome(...);
22     SignOnLocal signOn = home.create();
23     ...
24     signOn.createUser(userName, password);
25     ...
26   }
27 }
```

Figure 4.11: *Example of Pet Store Event-Handlers*

where it is mapped to its event-handling class of `HTMLAction` type, whose event-handling method is executed. If the result of the event-handling method produces an object of type `EJBEvent`, it is sent to the EJB tier, where it is mapped to its event-handling class of the `EJBAction` type, whose event-handling method is then executed.

Another major difference between GUI applications and J2EE applications is how the events are mapped to their corresponding event-handlers. While registration of the event-handlers in GUI applications happens statically, i.e., according to the setup code, registration of the event-handlers in J2EE applications happens dynamically, according to XML descriptor files, used at deployment time. Figure 4.10 shows excerpts of mappings.xml, distributed with the Pet Store example. This example shows that the `createUser` event is handled by `CreateUserHTMLAction` in the web tier and `CreateUserEJBAction` in the business tier.

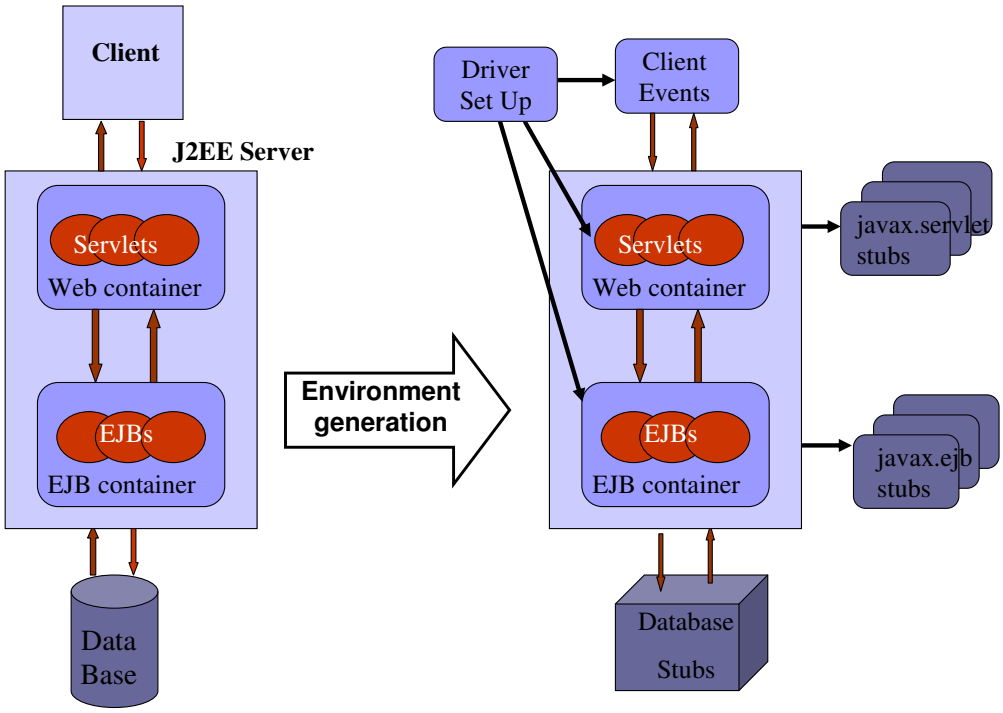

Figure 4.12: *Environment Generation for J2EE Applications*

Figure 4.11 shows excerpts from the code for the two event-handling classes used to process the `createUser` event. The `CreateUserHTMLAction` class implements the event-handling method `perform()`, which takes an argument of the `HttpServletRequest` type, retrieves user data from it, such as `userName` and `password`, and creates and object of type `CreateUserEvent`. The `createUserEJBAction` class also implements the event-handling method `perform()`, which takes the event produced by the `CreateUserHTMLAction` class, retrieves `userName` and `password` and invokes `createUser()` of `SignOnEJB`, using its local interface `SignOnLocal`.

4.2.3 Domain-Specific Methodology

Figure 4.12 shows our environment generation methodology for enterprise applications. In this domain, the Java part of a given enterprise application, excluding libraries, is treated as the unit under analysis. This part of the application-specific code contains implementation of servlets, EJBs, and event-handlers for the web and business tiers. Driver generation creates drivers that instantiate objects according to deployment descriptor files and simulate user actions on a browser. Stub generation replaces actual components such as a database and the J2EE libraries with Java implementations that preserve important behaviors but run in a non-distributed program.

Interface Discovery

As described in section 3.2, the general approach discovers the syntactic unit-environment interface. The unit interface in the general approach consists of all public methods and fields of the unit. However, in the J2EE domain, we are interested in the event-handling classes and their event-handling methods, which can be used to simulate user actions by the driver generation step.

Driver Generation

One approach to driver generation is to use the information about the screens and buttons available to the user at each step, similar to GUI applications. While we can get a handle on the GUI components, such as windows and buttons, we cannot do the same for J2EE applications, since such components are usually described by non-Java artifacts, e.g., HTML. One solution is to use HTML descriptions of the screen transition diagram for J2EE applications and automatically build the screen transition diagram in Java. We leave this approach for future work (section 8.2).

Another approach is to use BEG's support for regular expressions to describe actions of a user interacting with an application through a browser. When a user pushes a button or enters text into a text box, an `HttpServletRequest` is generated and sent to an event

handling class for processing. Driver generation produces a Java implementation of the user component that sets up the event-handling mechanism of J2EE applications during deployment. The driver contains two sections: setup and user actions. The setup section instantiates the event-handling classes of the web and business tiers and creates binding between the application events and the event-handlers according to deployment descriptor files. The user actions section reflects two aspects of the user behavior: sequences of the user actions and the user data. Both aspects are described using BEG's specification language. Specifically, we use regular expressions to describe sequences of actions; user data is described using concrete, choice, or abstract values.

To generate user inputs, we need to populate `HttpServletRequest` objects with keys, denoting their input source, and values denoting user inputs. Note the usage of `getParameter()` APIs, shown in Figure 4.11:

```
String userName = (String)request.getParameter("j_username");
String password = (String)request.getParameter("j_password");
String password2 = (String)request.getParameter("j_password_2");
```

These APIs are used to retrieve user inputs from the `HttServletRequest` objects. The keys, e.g., `"j_username"` denote the source of the user input, in this case, the text field where the user name must be entered. We configure BEG to track constant strings used in domain-specific APIs, e.g., in `getParameter()` calls. The set of such strings is finite and is used to generate code for population of `HttpServletRequest` with specific keys and their corresponding values.

Section 7.4 presents use case scenarios and drivers developed for the Pet Store example.

Stub Generation

Containment We configure BEG to track side-effects to the fields relevant to containment properties, e.g., the fields holding session information, context information, and user data. This information is used for development of initial stubs. Some initial stubs may require manual refinement, however, once developed for a specific framework, stubs can be reused across multiple applications belonging to the same framework.

```
1  public class HttpServletRequestImpl implements HttpServletRequest {
2    Hashtable params = new Hashtable();
3    ...
4    public void setParameter(String p1, String p2){
5      params.put(p1, p2);
6    }
7    public String getParameter(String p1){
8      return params.get(p1);
9    }
10   ...
11 }
```

Figure 4.13: *HttpServletRequest Stub*

Figure 4.13 shows excerpts from our stub implementation of the `HttpServletRequest` APIs, based on the analysis of the `MockHttpServletRequest` class implemented by the Spring Framework [31]. Implementation of the mocks objects by the Spring Framework uses `HashMaps` and `Hashtables` to implement containment properties. We configure BEG to track callbacks to such classes. Note that the field `params` is used to keep a mapping between keys denoting a source of user inputs and values denoting the user inputs.

Database Currently, we do not perform analysis of the database implementation. Because of the native code, it is a complex component to analyze. For the Pet Store example, we used BEG to generate empty stubs for the `java.sql` classes. Then we manually refined `ResultSetImpl`, the empty stub implementation of the `java.sql.ResultSet` APIs, to contain a two-dimensional array to hold data. This model can be reused across multiple applications. We note that for some properties, it is sufficient to model the database using empty stubs.

In this chapter, we studied two frameworks: Swing/AWT and J2EE. We identified their features that are relevant to checking properties related to their logical state, not their look and feel properties. The two domains share common features such as the event-handling mechanism, used to process user events and can be exploited to generate drivers. To generate stubs, we identified common important features, e.g., containment. These two domains

identified the need to customize the following BEG's features:

- Interface Discovery: BEG's unit interface discovery was customized to scan for special event-handling classes and their event-handling methods.
- Driver Generation: Generation of the setup section was customized to take into account the GUI setup for GUI applications and mappings from event to their event handlers for J2EE applications. In addition, we developed a universal driver for GUI applications which, on top of the application-specific setup, can be reused across different GUI applications.
- Stub Generation: BEG's side-effects analysis was customized to track side-effects to fields that implement domain-specific features, e.g., containment.

In the next chapter, we describe our environment generation techniques. We describe the algorithms used for interface discovery, BEG's grammar for user specifications, formally present BEG's side-effects analysis, and describe BEG's code generation techniques.

Chapter 5

Environment Generation Techniques

In this Chapter, we present our environment generation techniques. Specifically, we focus on (1) automatic discovery of the unit-environment interface, (2) BEG's specification language, (3) BEG's static analysis for extracting assumptions, and (4) code generation techniques for encoding environment assumptions into Java code. We describe each of these in turn and conclude with limitations of the current approach and possible extensions.

5.1 Program Representation

Interface discovery and static analysis techniques in BEG work on Jimple, a three-address representation of JVM byte-codes used in the Soot framework [91]. We consider the following syntactic categories: classes $c \in Class$, methods $m \in Method$, fields $f \in Field$, statements $s \in Stmt$, expressions $e \in Expr$, local variables $p, l, r \in Var$ (with p denoting parameters and l/r used on the left/right hand side of assignment expressions), operators $op \in Operator$, and types $t \in Type$.

We assume presence of operators for relating fields and methods to their containing class, similar to Java reflection APIs and methods implemented in Soot, $f.getDeclaringClass()$, $m.getDeclaringClass()$, and operators relating classes to their methods and fields, $c.getMethods()$ and $c.getFields()$. We assume that all expressions in our language have types and that the type of each expression e can be found using the function $e.getType()$. We assume that an expression e of scalar type can be evaluated to a constant if a function $e.isConst()$ returns

`true`. Since the programs we analyze are precompiled, we assume that they contain no typing errors, for example, all field access expressions are type correct, conditionals for `if` and `while` statements have a boolean type (which is included in the scalar type), and the terms in the expression $e_1\ op\ e_2$ have types for which op is defined.

We denote the classes identified as the unit as $U \subseteq Class$. For convenience, we use f_U to denote the set of fields where $f.getType() \in U$.

5.2 Interface Discovery

There are two aspects of the unit-environment interface that BEG addresses: actions that the environment may perform on the unit and actions that the unit may perform on the environment. BEG automatically discovers both, which we describe next.

5.2.1 Unit Interface

BEG walks over unit classes to discover the unit interface, which, in general, consists of public methods and fields of the unit classes. Based on a specific domain, the entry points into the unit under analysis can be restricted. For example, as described in Chapter 4, for GUI and web applications, these are defined as special event-handling methods that process user inputs such as button clicks and writing into a text field.

Algorithm 1 shows our algorithm for discovering the unit interface. Methods `isRelevantMethod()` and `isRelevantField()` define domain-specific information about methods and fields that may be exercised by unit drivers. The set of discovered entry points is used to build an alphabet of default environment actions, which consist of field assignments and method calls, including constructor calls. The unit interface information is used to validate user assumptions and, if no assumptions are provided, the default actions are used to generate universal drivers.

Algorithm 1 Unit Interface Discovery

Input: U: set of unit classes
Output: S: set of unit methods and fields
Initially: $S = \emptyset$

```
1: for each class u ∈ U do
2:    for each method m ∈ u.getMethods() do
3:       if isRelevant(m) then
4:          S = S ∪ m
5:       end if
6:    end for
7:    for each field f ∈ u.getFields() do
8:       if isRelevant(f) then
9:          S = S ∪ f
10:      end if
11:   end for
12: end for
13: return S
```

5.2.2 Environment Interface

BEG walks over unit classes to discover external references, which consist of classes, methods, and fields directly referenced by the unit. The external references are used to define boundaries of the needed environment. They are also used to validate user assumptions for stubs and to build empty and universal stubs, if no assumptions are given.

Algorithm 2 shows the algorithm for discovering all external references of a given unit. For each of the unit classes, the algorithm checks whether its parent or any of the interfaces it implements need to be generated in the environment. Next, the algorithm walks over fields and methods of each unit class. For each field, it checks whether its type needs to be created in the environment. For each method, it checks its signature and analyzes the list of locals and statements. The method `checkMethodSignature(m)` checks the return type, the types of parameters and possible exceptions. Statements are analyzed for external method calls and external field references. The call graph is used to resolve virtual invoke expressions. The method `checkEnv(c)` checks whether class `c` is external to the unit, and if it not already in the environment, it adds that class to E.

Algorithm 2 Environment Interface Discovery

Input: : U: set of unit classes, CG: call graph
Output: : E: set of environment classes, methods, and fields
Initially: $E = \emptyset$

```
1: for each class u ∈ U do
2:   envCheck(u.getSuperclass())
3:   for each interface I ∈ u.getInterfaces() do
4:     envCheck(I)
5:   end for
6:   for each field f ∈ u.getFields() do
7:     envCheck(f.getTypeClass())
8:   end for
9:   for each method m ∈ u.getMethods() do
10:    envCheckSignature(m)
11:    for each local l ∈ m.getLocals() do
12:      envCheck(l.getTypeClass())
13:    end for
14:    for each statement s ∈ m.getStatements() do
15:      if (s.containsInvokeExpr()) then
16:        method m' = CG.resolveDispatch(s)
17:        envCheckSignature(m')
18:        class D = m'.getDeclaringClass()
19:        if (envCheck(D)) then
20:          E = E ∪ m'
21:        end if
22:      end if
23:      if (s.containsFieldRef()) then
24:        f = s.getFieldRef()
25:        class D = f.getDeclaringClass()
26:        if (envCheck(D)) then
27:          E = E ∪ f
28:        end if
29:      end if
30:    end for
31:  end for
32: end for
33: return E
```

$$
\begin{array}{rcl}
T & ::= & \text{boolean} \mid \text{int} \mid C \\
action & ::= & T\ l \mid l := e \mid l.f := e \mid l.m(\bar{l}) \\
e & ::= & p \mid C.f \mid r \mid r.f \mid \text{null} \mid a \mid r.m(\bar{r}) \mid \ \text{new}\ C(\bar{r})
\end{array}
$$

Figure 5.1: *Action Syntax*

Note that when discovering driver actions, domain-specific information can be used to constrain the set of actions. When discovering environment interface, *all* external references are added, regardless of domain-specific information, to make the unit compilable with the set of produced stubs. The domain-specific information about stubs is used later, when discovering their behavior.

5.3 Specifying Assumptions

In this section, we describe BEG's specification language. We focus on BEG's support for specifying program *actions* and *patterns* of actions that the environment may perform on the unit. The same actions and patterns of actions apply to both drivers and stubs. In case of drivers, the patterns of actions reflect implementation of methods belonging to driver threads (e.g., `main()` and `run()`), whereas in case of stubs they reflect implementation of non-driver methods.

5.3.1 Specifying Actions

Environment may perform various actions on the unit, e.g., call unit's methods, perform assignments to unit fields, instantiate unit classes. BEG specification language has support for describing such actions. Figure 5.1 shows BEG's actions simplified syntax, which allows for specification of variable declarations and many Java expressions, including assignments, method invocations, and object allocations. Barred identifiers indicate finite lists, for example, $\bar{l}$ denotes a list of variable names. In general, any Java expression or statement can

be used to describe a single environment action as long as there is no interference with the special logical operators used to describe patterns of actions.

In addition, BEG recognizes *abstract* and *choice* modeling primitives declared by `Verify`, `Bandera`, and `Abstraction` classes described in section 2.1.6. These modeling primitives fit into the action syntax, since they are either field reference expressions or method call expressions. Next, we describe syntax of field assignments and method call actions in more detail.

Field assignments can be either static field assignments or assignments through object references of unit type. Assignments are of the form $l.f = r$ where: $type(l) \in U$, f is of scalar or unit type, and r is either a concrete value (e.g., a scalar constant), *choice*, or *abstract* value. The target of the assignment, l, is a previously introduced local, *choice* or *abstract* value of appropriate type, or the name of a class when a static field assignment is to be specified.

Method call actions are defined using standard Java syntax, but where partial specification of parameters is allowed. Missing parameters in a method call are interpreted as actual parameters with concrete, *abstract*, or *choice* value for the formal parameter type. BEG can be configured to emit any of the three types of values. For example, consider several methods named `m` in class C with signatures:

```
public R m(P x, Q y) { ... }  // method 1
public R m(P x) { ... }  // method 2
public R m() { ... }  // method 3
```

We can denote the occurrence of a call to the first method with any receiver object of type C, a specific value, p_1, for `x`, and a choice value for `y` as `m(`p_1`, chooseClass("Q"))`. The meaning of such an action is the nondeterministic choice of a call on method 1 from the set of all calls that can be constructed by selecting instances of C and Q for the receiver and `y` parameter, respectively, and using p_1 for parameter `x`. To denote the occurrence of a call to any of a subset of methods named `m`, one can elide the parameter list suffix that distinguishes the elements of the set. For example, `m(`p_1`)` denotes a call to either method 1

$$
\begin{array}{rcl}
spec & :== & action \\
 & :== & spec; spec \\
 & :== & spec|spec \\
 & :== & (spec) \\
 & :== & spec? \\
 & :== & spec* \\
 & :== & spec+ \\
 & :== & spec + \{n\} \\
 & :== & spec + \{n, m\}
\end{array}
$$

Figure 5.2: *Regular Expressions Assumptions Syntax*

or 2 with any receiver object and p_1 for parameter `x`. The action `m()` denotes a call to any of the three methods with any parameter values.

In the absence of a driver specification, BEG uses information about the unit interface, consisting of unit fields and methods, and constructs *default* driver actions consisting of field assignments and method calls with parameter values elided. The elided values get filled in during the code generation step.

5.3.2 Specifying Patterns of Actions

Different formal notations can be used for specifying patterns of program actions. Currently, BEG has support for LTL and regular expressions. LTL is usually supported by model checkers and is commonly used to specify properties. One can use LTL to specify environment assumptions and, when the environment code is available, the same assumptions can be used to check that the environment satisfies its LTL properties. Our approach to LTL specifications is similar to the one developed for model checking Ada programs by Pasareanu et al. described in [66].

Regular expressions are a familiar formal notation to many developers and our experience shows that many find it easier to use than temporal logics. Next, we describe our approach to regular expressions specifications in detail.

Regular expressions defined over the alphabet of program actions describe a language of actions that can be initiated by the environment. The simplest regular expression is

a single program action. Complex environment assumptions are built up using the standard operators for regular expressions: ; (concatenation), | (disjunction), $*$ (closure), and ? (optional occurrence). Positive closure (+), bounded iteration ($r + \{n\}$ the concatenation of n occurrences of r), and a generalization of bounded iteration ($r + \{n, m\} = r + \{n\}|r + \{n + 1\}| \ldots |r + \{m\}$, where n and m are a pair of ordered natural numbers) are also supported. The syntax of these assumption specifications, *spec*, is given in Figure 5.2, where *action* is a program action.

As an example, `java.util.Iterator` presents a standard interface for generating the elements in an instance of a container. Semantically, this interface assumes that for each instance of a class implementing the `Iterator` interface (denoted by the introduced name `i`), all clients will call methods in an order that is consistent with the following specification:

```
Iterator i = iterator();
(i.hasNext(); i.next(); i.remove()?)*
```

This expresses both required sequencing of calls (e.g., a call to `iterator()` on some instance of a class that implements the `Iterator` interface must precede a call to `hasNext()`) and allowable optional calls (e.g., the occurrence of a single `remove()` call after a call to `next()`) over each instance of `Iterator`.

In the absence of user specifications, BEG can be configured to generate universal environments. During the interface discovery step, BEG finds all methods and fields in the unit interface and, using them, constructs default field assignments and method calls. Suppose the alphabet of default driver actions is $a_1 \ldots a_n$, then the universal environment will correspond to a regular expression $(a_1| \ldots |a_n)*$.

5.3.3 Specifying Drivers and Stubs

Once the patterns of actions are specified, the user needs to specify which environment methods implement them. Figure 5.3 shows the syntax of BEG's specification. There are several section in BEG's specification file: definitions, setup, and assumptions. The optional *definitions* section can be used to give mnemonics to hard-to-read actions. We use

```
environment {
    (definitions{(key = value)*})?
    (setup{(num Type;)*})?
    (driver-assumptions{
        ( re |ltl { Main | num? ThreadClassName: spec #} )*
    })?

    (stub-assumptions{
        (ClassName{
             ( re|ltl MethodSignature{ spec (return val)?})*
        })*
    })?
}
```

Figure 5.3: *Driver and Stub Assumptions Syntax*

this feature in section 7.1 to give easy-to-understand names to user actions. We also use this feature to specify a mapping from events to their event handlers for J2EE applications, as described in section 7.4. The optional *setup* section can be used to specify the number of instances per unit type the main method should create. We use this feature to specify the setup for the observer-observable example in section 3.1. More complicated setup behavior can be specified in the assumptions section, using `Main` as the name of the driver thread.

The assumptions section may contain specifications for drivers and stubs. Note that the user can mix LTL and regular expressions descriptions in one specification: some driver threads and stub methods can be described using LTL, others using regular expressions. In the driver-assumptions section, the user describes the behavior of the `main` method of the main thread, using `Main` as the thread name, and the `run` methods of the other driver threads. Specification of the `Main` thread describes the behavior of the main thread before the other threads are started. In addition, we can have support for describing the behavior of the main thread after the other driver threads are done, e.g., by supporting a *cleanup* section. However, in our case studies, we did not have the need for the clean up section.

In the stub-assumptions section, the user may specify some stub methods; the rest of the stubs are automatically generated by BEG. Specification of stub methods is similar to specification of driver methods, except for an optional `return val` statement. The user

may also skip specification of the return statement and let BEG fill it with a value of the appropriate type.

5.4 Extracting Assumptions

In this section, we discuss static analysis techniques used in BEG to extract certain environment assumptions. In general, interactions between the unit and its environment can be complicated and difficult to analyze: the environment may influence the unit's *data* (e.g., by modifying objects flowing from the unit) and *control* (e.g., by invoking various methods in the unit's interface). There are techniques such as *slicing*, which can calculate all possible dependencies between the unit and its environment, however, slicing techniques can be expensive. Static analyses targeted to calculate only certain aspects of unit-environment interactions can be more effective and produce more compact Java models. Based on our experience with case studies stubs are usually passive components with few callbacks and synchronization, therefore, we mainly concentrate on calculating data dependencies. Specifically, we describe side-effects analysis, which calculates a set of objects possibly modified by a method, and points-to analysis, which for each reference variable calculates a set of objects that the reference may point to. The points-to analysis is a prerequisite for side-effects analysis.

Next, we present our points-to and side-effects analysis, described in detail in [87, 85]. To perform case studies presented in this thesis, we needed to make it extensible and tune it to different domains. As a result we implemented extensions that track containment and features specific to GUI and J2EE libraries. These extensions do not change the algorithm that calculates points-to and side-effects information. Only the definition of what constitutes the unit under analysis needs to reflect domain-specific information. This is done using extensible APIs in BEG, which we describe in chapter 6.

As mentioned in section 2.1.3, we use Data Flow Analysis (DFA) to perform static analysis. Note that a DFA is fully defined by the lattice of data values propagated through

CFG, the transfer functions, the combination operator, the direction of propagation, and the initial data values.

Points-to and side-effects analysis results are only dependent on assignment statements and method calls. Assignments in three-address form, `l.f = r`, always refer to a local reference-type variable `l` in forming the target address; due to the 3-address form, we do not need to consider complex dereference expressions on the left-hand side of assignments; due to reference type of `l`, the points-to analysis keeps track only of non-scalar type variables; assignments to scalar-type variables are ignored. We restrict our analysis presentation to the following statements:

identity: `l = p`

copy: `l = r`

load: `l = r.f`

store: `l.f = r`

allocation: `l = new C`

invoke expression: $\texttt{r}_0\texttt{.m(r}_1\texttt{, ...,r}_n\texttt{)}$

invoke statement: $\texttt{l = r}_0\texttt{.m(r}_1\texttt{, ...,r}_n\texttt{)}$

Our analyses treat array access expressions similarly to field access expressions and non-virtual invoke statements similarly to non-virtual invokes; for simplicity we restrict our language and limit the presentation to reference and scalar types and to virtual invokes. For details on other statements, please refer to [85].

5.4.1 Abstract Access Paths

Fundamental to our analyses is our approach for representing the memory locations that a statement may reference. Our approach is based on length-limited access path based analyses (e.g., [51]) and parameterized analyses (e.g., [54]). We combine these approaches

```
1  class Node{
2    Node next;
3    Data data;
4    ...
5  }
6
7  class ...{
8    void m(Node n, Data d){
9      n.next.next.data = d
10   }
11 }
```

```
1  //Jimple representation:
2
3  class ...{
4    void m(Node n, Data d){
5      ...
6      t = n.next;
7      t = t.next;
8      t.data = d;
9    }
10 }
```

Figure 5.4: *Example in Java and Jimple to Demonstrate Naming of Access Paths*

and adapt them to our setting in which the analysis distinguishes between unit data and environment data to precisely characterize points-to information for the former, but not the latter.

The goal of our analysis is different from the traditional goal of points-to analyses. In particular, we do not use analysis results to determine potential aliasing relationships and therefore do not require a *canonical* representation of points-to information. Our analyses are used to safely represent, at each program point, the state of the program, which maps variables to their values; values can be heap locations, a special value *null* denoting a null pointer, or scalar values (e.g., integers, reals, etc.).

An important part of an access path based points-to analysis is coming up with a scheme for access paths used to name heap objects. To give the intuition behind our access paths names, let us look at a simple example shown in Figure 5.4; the right side shows the example in Java, the left side shows the code using the Jimple representation. Class `Node` shown in the example serves as a list where `Data` objects may be stored. Suppose, types `Node` and `Data` are unit types and suppose method `m` is an environment method under analysis. Let us trace what happens to unit data being passed to the method `m`.

Tracing Assignments Through the Concrete Heap For demonstration purposes, suppose that before the first assignment of method `m` (in Jimple) is executed, the heap

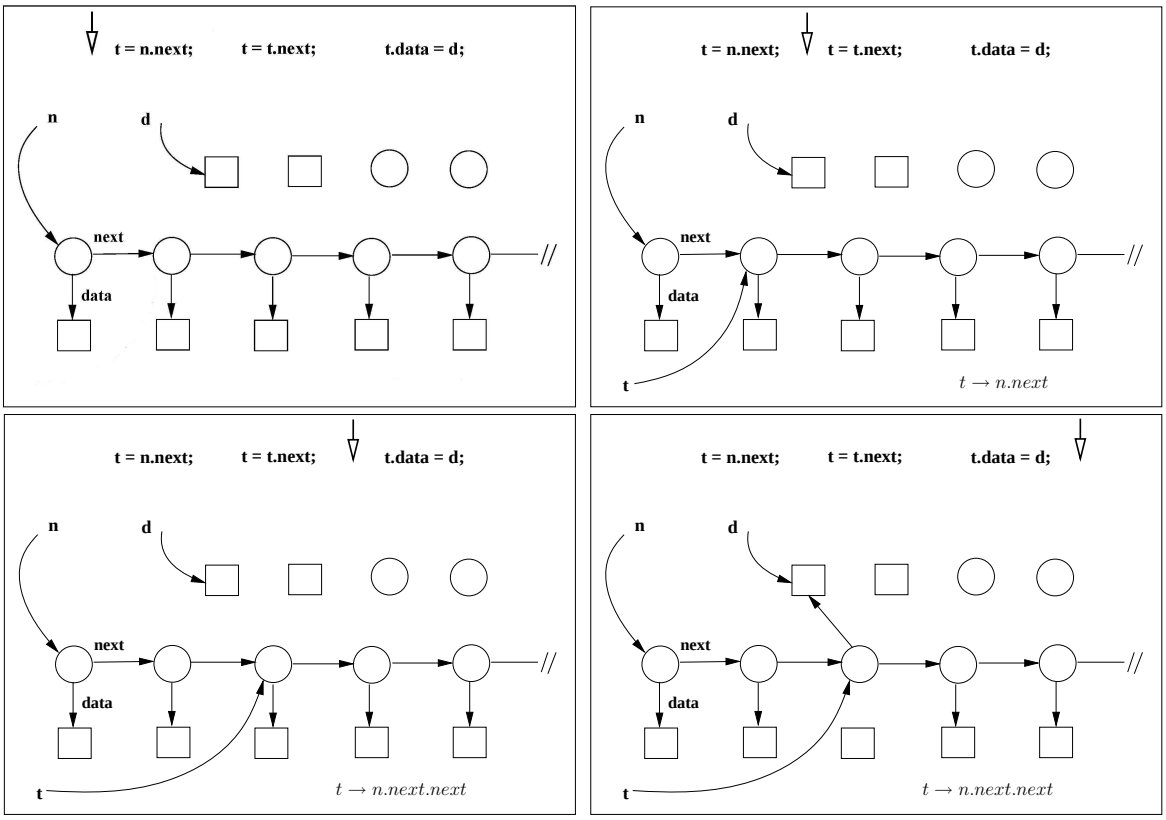

Figure 5.5: *Tracing Assignments Through the Concrete Heap*

looks as shown in the upper left picture of Figure 5.5. Then, the next three pictures show the heap after executing the first, second, and third assignments. It is easy to see that at the end of the method execution the variable `t` points to a heap node that may be named *n.next.next*. However, in general, chains of references through the heap may be infinite and cannot be statically bound to heap objects. One solution to this problem is to use a *k-limiting* bound on the length of access paths. Let us trace the example using an analysis with 1 as a k-limiting constant.

Tracing Assignments Using 1-Limited Analysis Figure 5.6 (left) shows the heap after execution of the first assignment. The picture is the same as before except now we are using dashed arrows rather than solid ones to show relationships between reference variables and heap objects. This is done to emphasize that according to the analysis results

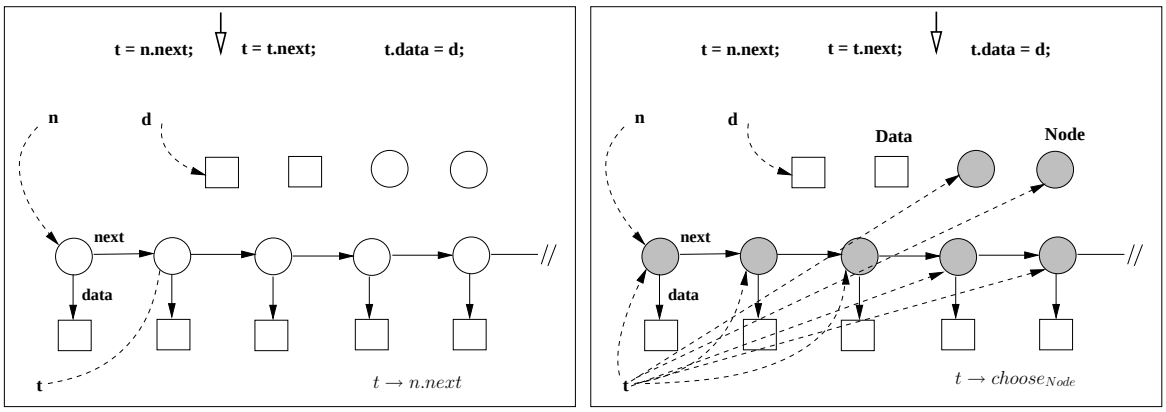

Figure 5.6: *Tracing Assignments Using 1-Limited Analysis*

the variable `t` *may* point to a node named $n.next$. After the second assignment is executed, `t` may point to the node that can be named $n.next.next$, beyond k-limit. Usually, once the k-limiting bound is reached, the analyses loose some or all information about heap locations being described. However, in Java, we can exploit the type information and trace that after execution of the second assignment, the variable `t` may point to a heap object of type `Node` (not to *any* heap object). The right side of Figure 5.6 shows this by having `t` point to any heap object (shaded) of type `Node`. We use a special name $choose_{Node}$ to denote a set of all heap objects of type `Node` at a given program state.

Note that this analysis yields very imprecise results: after the execution of the third statement, the `data` field of every shaded box may point to `d`. To improve the precision of this analysis, we track reachability of heap objects.

Tracing Assignments Using 1-Limited Analysis with Reachability Figure 5.7 shows steps for the analysis improved with reachability tracing. The heap is shown as before after the execution of the first assignment. However, after the execution of the second assignment, this analysis can distinguish between all heap objects of type `Node` from the heap objects of type `Node` that are also reachable from the node $n.next$ through heap references. The shaded objects on the right side of Figure 5.7 show the set of `Node` objects that `t` may point to after the execution of the second assignment. We use a special name

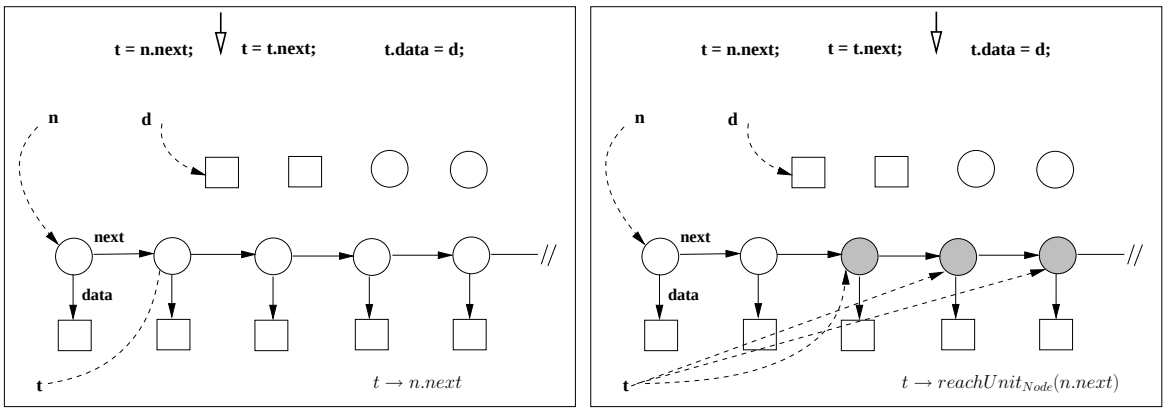

Figure 5.7: *Tracing Assignments Using 1-Limited Analysis with Reachability*

$reachUnit_{Node}(n.next)$ to denote this set.

This example gives the intuition behind exploiting the type and reachability information for heap objects. Note that the access paths explained in the example started with `n`, the name of the formal parameter. In general, access paths may start with a name for an object of unit type that an environment method may modify. There are several categories of such objects: globals, parameters, and newly created objects. It is convenient to set up a symbolic location for each of them: global locations, denoted by $C.f$, parameter locations, denoted by p_i, with p_0 representing $this$, and newly allocated objects, denoted by $new_{C,s}$, for all instances of c created by statement s.

Note that object may flow to the environment through the return value of called methods. For each object returned from a method, we can trace whether the object is a parameter, global, or newly created object. The above symbolic locations are called roots. The naming scheme for roots guarantees uniqueness of their names within a method.

Formal Discussion

More formally, our points-to representation captures information relative to a given method and is parameterized by a *root* symbol that represents memory locations. There are three kinds of memory locations that may serve as a root: public static fields of classes (denoted $C.f$), method parameters (denoted p_i), and newly allocated data (denoted $new_{C,s}$ for objects

of type C allocated at statement s). New locations are modeled as per-allocator summary locations which are supported by the environment code generation described in Section 5.5. Our representation makes use of operations that denote sets of heap-allocated objects in a given state. One can access the set of all allocated instances of class C (denoted $choose_C$), and the set of allocated instances of class C that are reachable from memory location l via paths through the heap that only reference unit data (denoted $reachUnit_C(l)$).

A memory location holds a value that is a scalar, a *null*, or a heap object. To represent non-scalar memory locations tracked by the points-to analysis, we denote them by an *abstract access path* $\pi \in AbsPath$ that may be defined by the following regular expression:

$$\pi \quad = \quad [(C.f \mid p_i \mid new_{C,s}) f_U^{0-k} (reachUnit_C)?] \mid choose_C \mid null$$

An access path is either $choose_C$ expression, a *null*, or a length-limited path that starts at a root symbol, consists of 0 to k dereferences of field accessors of unit type, and is optionally terminated in a reachable expression. Alternatively, we use notation $reachUnit_C(\pi)$ (where the parameter is understood to be the path prefix) to denote $(\pi)reachUnit_C$. We refer to a prefix of a path with j field dereferences as $\pi[j]$. The semantics of a path are defined relative to a program state. Paths represent field accesses that are *type correct* in the sense that $\pi[j]$ with type C can only be extended to length $\pi[j+1]$ by a field f where $class(f)$ is C. Length-limited paths end in either the location referred to by the field access sequence or a $reachUnit_C(\pi[k])$ expression. In the former case, the access path represents instances of class C that are reachable via the chain of field dereferences denoted by π in state s. In the latter case, the access path represents instances of class C that are reachable via a chain of field dereferences through unit data from any of the memory locations denoted by $\pi[k]$ in state s. Note that a variable, $l \in Var$, of a reference type may point to a set of locations, $\Pi \in \mathcal{P}(AbsPath)$, whose types are assignment-compatible (i.e., $\pi \in \Pi \Rightarrow type(\pi) \leq type(l)$, where $\leq$ is the sub-typing relation).

Our abstract access paths provide a different degree of precision compared to traditional k-limited access path based representations (e.g., [51]) in that they are well-typed and they

are able to represent heap reachability relationships between locations.

A pair of abstract access paths is ordered ($\leq$) based on the containment order of the sets of memory locations denoted by the pair. According to the semantics described above the order is:

$$\forall_{j \leq i} \qquad \pi[i] \leq reachUnit_{type(\pi[i])}(\pi[j])$$
$$\forall_{C,i} \qquad reachUnit_C(\pi[i]) \leq choose_C$$

This ordering is lifted to sets of symbolic locations as follows:

$$(\forall_{a \in S} \exists_{b \in S'} a \leq b) \rightarrow S \leq S'$$

An abstract access path can be *extended* by a field dereference (denoted $\pi.f$) using rules that distinguish unit locations from environment locations. If $\pi.f$ refers to an environment location, the analysis does not keep track of it:

$$type(f) \notin U \implies \pi.f = choose_{type(f)}$$

Otherwise, if $\pi.f$ refers to a unit location, we must consider the structure of π:

$$type(f) \in U \implies \pi.f = \begin{cases} (root)f_U^i f & \text{if } \pi = (root)f_U^i \wedge i < k \\ reachUnit_{type(f)}(\pi) & \text{if } \pi = (root)f_U^k \\ reachUnit_{type(f)}(\pi') & \text{if } \pi = reachUnit_C(\pi') \\ choose_{type(f)} & \text{if } \pi = choose_C \end{cases}$$

An abstract access path rooted in a formal parameter $\pi(p)$ can be *prefixed* (denoted $\pi(\pi'/p)$) by substituting the name of formal parameter p, used in defining access path, with type and length appropriate path prefix π'. If a prefix operation causes the sequence of field dereferences to exceed k then the extension operator is applied for each field dereference beyond k. The intuition here is that we calculate a symbolic analysis summary for a method m and then use the prefixing operator (denoted $m(\bar{\pi}/\bar{p})$) to determine the effects at a call site by substituting the actual parameters, represented by $\bar{\pi}$, for the symbols representing the formal parameters, $\bar{p}$, of m.

Extension and prefixing operations can be lifted to sets of abstract access paths $\Pi \in \mathcal{P}(AbsPath)$ by extending or prefixing each constituent path (denoted $\Pi.f$ and $\Pi(\bar{\pi}/\bar{p})$).

5.4.2 Points-to Analysis

Our points-to analysis is a flow-sensitive, forward-flow analysis. A set of points-to mappings from method reference-type locals to sets of abstract locations that they may/must point to

$$Pt^{may}, Pt^{must} : Var \rightarrow \mathcal{P}(AbsPath)$$

is calculated for entry and exit of each statement in the flow graph. An additional mapping

$$Pt_m^{may}, Pt_m^{must} : Method \rightarrow \mathcal{P}(AbsPath)$$

maps methods to their return locations as calculated by the points-to analysis at the exit point of the methods.

The lattice of the may analysis is a powerset lattice $L = (\mathcal{P}(Var \rightarrow \mathcal{P}(AbsPath))$ with partial order $\subseteq$, least upper bound operator $\cup$, and least element $\emptyset$. The lattice of the must analysis is a powerset lattice $L = (\mathcal{P}(Var \rightarrow \mathcal{P}(AbsPath))$ with partial order $\supseteq$, least upper bound operator $\cap$, and least element $Var \rightarrow \mathcal{P}(AbsPath)$. The initial data flow set is empty for both analyses. Sets are combined at flow-graph merge points by unioning the images of mappings with the same domain element for may analysis and intersecting them for must analysis.

The data flow equations are defined as follows:

$$\begin{aligned}
Pt_{entry}^{may}(s) &= \bigcup \{Pt_{exit}^{may}(s') \mid s' \in pred(s)\} \\
Pt_{exit}^{may}(s) &= (Pt_{entry}^{may}(s) - Kill(s)) \cup Gen(s) \\
Pt_{entry}^{must}(s) &= \bigcap \{Pt_{exit}^{must}(s') \mid s' \in pred(s)\} \\
Pt_{exit}^{must}(s) &= (Pt_{entry}^{must}(s) - Kill(s)) \cup Gen(s)
\end{aligned}$$

To simplify presentation, we define operation $Locs(l, s) = \{\pi \mid (l \rightarrow \pi) \in Pt_{entry}(s)\}$, that given the local variable `l` and the statement s, returns the set of locations that l

may/must point to at the entry point of s; to further simplify the notation, we omit the second parameter s, but it is understood that $Locs(l)$ is evaluated using the may/must points-to information at the entry point of statement under analysis.

We define transfer functions for assignment statements s shown in Figure **??** in terms of $Kill/Gen$ functions. For assignment statements, the $Kill$ sets are of the form (to avoid specifying two sets of functions, we omit *may/must* superscript for $Pt_{entry}(s)$):

$$\begin{aligned} Kill(\texttt{l = ...}) &= \{l \rightarrow \Pi \mid (l \rightarrow \Pi) \in Pt_{entry}(s)\} \\ Kill(\texttt{l.f = r}) &= \{l' \rightarrow \pi \mid (l' \rightarrow \pi) \in Pt_{entry}(s) \wedge \\ &\quad \exists i \mid \pi[i] \in Locs(l).f\} \end{aligned}$$

The $Kill$ function for the store statement calculates all references l' that may/must point to a location whose access paths π contains the heap reference f that gets modified by the statement. As a safe approximation, such variables will point to $choose_{type(l')}$ after the statement.

For statements whose assigned type is not in the unit the Gen set is:

$$Gen(\texttt{l = ...}) = \{l \rightarrow choose_{type(l)}\}$$

In all other cases the Gen sets for statement s are:

$$\begin{aligned} Gen(\texttt{l = p}) &= \{l \rightarrow \{p\}\} \\ Gen(\texttt{l = r}) &= \{l \rightarrow Locs(r)\} \\ Gen(\texttt{l = r.f}) &= \{l \rightarrow Locs(r).f\} \\ Gen(\texttt{l.f = r}) &= \{l' \rightarrow choose_{type(l')} \mid (l' \rightarrow \pi) \in Kill(s)\} \\ Gen(\texttt{s: l = new C}) &= \{l \rightarrow \{new_{C,s}\}\} \\ Gen(\texttt{l = r}_0\texttt{.m(r}_1,\ldots\texttt{)}) &= \{l \rightarrow Pt_m(m)(\bar{Locs}(r_i)/\bar{p_i})\} \end{aligned}$$

$Gen(\texttt{l = (C)r})$ is defined using operation $setType(\pi, C)$. The operation modifies the type of abstract access path π to C, e.g., $setType(choose_{Object}, MyData) = choose_{MyData}$.

This operation is lifted to sets of abstract access paths by modifying the type of each constituent path.

$Gen($`l = r0.m(r1,...)` is defined using $Pt_m(m)$ which is a set of return locations as calculated by the points-to analysis for m. Return locations described by abstract access paths rooted in a formal parameter $\pi(p_i)$ are prefixed with the abstract access paths that describe locations of the corresponding actual parameter r_i; prefixing of $\pi(p_i)$ with locations of r_i is denoted $\pi(Locs(r_i)/p_i)$. Applying prefixing to all return locations in $Pt_m(m)$, we get the $Pt_m(m)(\bar{Locs}(r_i)/\bar{p_i})$ used in the definition of the Gen function.

5.4.3 Side-Effects Analysis

Side effects occur in **store** statements `l.f = r`. Our side-effects analysis uses the symbolic locations calculated for `l` at an assignment statement to determine the set of objects whose fields may be referenced as the target of the assignment. The value of the right-hand side of such an assignment is also safely approximated by looking up the abstract values referenced by `r`.

As mentioned previously, we calculate both *may* and *must* side-effects information. These are flow-sensitive, forward flow analyses. The analyses relate side-effected symbolic locations to sets of *abstract values*

$$AbsValue = \{AbsPath \cup Scalar^{\top}\}$$

Note that $Scalar^{\top}$ is the domain of values for all non-reference variables lifted to contain a $\top_t$ value, for each type t, that represent all possible values of type t; the values in $Scalar^{\top}$ are similar to values in a constant propagation lattice [58], however, our analysis can keep track of a set of constant values. An abstract value $v_a \in AbsValue$ such that $v_a = \pi(p)$ can be prefixed (denoted $\pi(\pi'/p)$). Abstract value prefixing is defined analogously to prefixing for abstract access paths; prefixing for scalar type values or access paths that do not originate in a formal parameter produces no change. Prefixing operation can be lifted to sets of abstract values $V \in \mathcal{P}(AbsValue)$ by prefixing each constituent value (denoted $V(\bar{\pi}/\bar{p})$).

A set of side-effects mappings from abstract locations to sets of abstract values

$$Se^{may}, Se^{must} : AbsPath \rightarrow \mathcal{P}(AbsValue)$$

is calculated for entry and exit of each statement in the flow graph. The lattice of the may analysis is a powerset lattice $L = (\mathcal{P}(AbsPath \rightarrow \mathcal{P}(AbsValue))$ with partial order $\subseteq$, least upper bound operator $\cup$, and least element $\emptyset$. The lattice of the must analysis is a powerset lattice $L = (\mathcal{P}(AbsPath \rightarrow \mathcal{P}(AbsValue))$ with partial order $\supseteq$, least upper bound operator $\cap$, and least element $AbsPath \rightarrow \mathcal{P}(AbsValue)$. Sets are combined at flow-graph merge points by unioning domain values with the same $AbsPath$ elements for *may* analysis and by intersecting them for *must* analysis. The initial data flow sets for *may* and *must* analyses are empty.

Transfer functions are defined for **store** and **invoke** statements as follows:

$$\begin{aligned}
Se^{may}_{entry}(s) &= \bigcup\{Se^{may}_{exit}(s') \mid s' \in pred(s)\} \\
Se^{may}_{exit}(s) &= (Se^{may}_{entry}(s) - Kill(s)) \cup Gen(s) \\
Se^{must}_{entry}(s) &= \bigcap\{Se^{must}_{exit}(s') \mid s' \in pred(s)\} \\
Se^{must}_{exit}(s) &= (Se^{must}_{entry}(s) - Kill(s)) \cup Gen(s)
\end{aligned}$$

For clarity we define $Vals(r, s) = Locs(r, s) \cup Scalars(r, s)$, where $Scalars(r, s)$ is defined for a scalar type r and returns a set of abstract scalar values r may hold at the entry point of statement s. For simplicity we omit specifying the second parameter s from these operations. $Scalar(r)$ is evaluated to a scalar constant if $isConst(r) = true$ and is evaluated to $\top_t$ otherwise.

For **store** statements where l may point to at most one memory location we can overwrite previous values of $l.f$, i.e., perform a strong update. The $Kill$ set in such a case is defined as follows (to avoid presentation of two sets of transfer functions we omit specifying superscripts

may/must for $Se_{entry}(s)$):

$$\begin{aligned} Kill(\texttt{l.f = r}) \;=\; & \{\pi.f \rightarrow V \mid \pi \in Locs(l) \wedge \\ & (\pi.f \rightarrow V) \in Se_{entry}(s) \wedge isSingular(l)\} \end{aligned}$$

If l may point to more than one memory location, overwriting previous values of $l.f$ may be unsafe. For such cases the $Kill$ set is empty; this is a weak update, which keeps previous values of `l.f`.

Gen sets for **store** and **invoke** statements, s, are defined as:

$$\begin{aligned} Gen(\texttt{l.f = r}) \;&=\; \{\pi.f \rightarrow Vals(r) \mid \pi \in Locs(l)\} \\ Gen(\texttt{r}_0\texttt{.m(r}_1, \ldots\texttt{)}) \;&=\; \{\pi(\bar{Locs}(r_i)/\bar{p_i}) \rightarrow V(\bar{Locs}(r_i)/\bar{p_i}) \mid \\ & \quad (\pi \rightarrow V) \in Se_m)\} \end{aligned}$$

where $Locs(l)$ denotes a set of locations l may point to according to either must or may points-to analysis; Se_m denotes either the must or may analysis summary for m. For all other statements the identity transfer function is used.

5.4.4 Analyzing Swing/AWT and J2EE components

As discussed in sections 4.1 and 4.2, when analyzing Swing/AWT and J2EE library classes, we want to keep track of side-effects to specific fields in the environment. The algorithm to calculate points-to and side-effects analyses does not need to change, however, the definition of the unit has to reflect domain-specific information.

In addition to using side-effects analysis to track domain-specific features like visibility, enabledness, and containment, we also found it useful to have automated support for generation of user events, a feature common for both GUI and J2EE applications.

Figure 5.8 shows `actionPerformed()` method implementation from the `ButtonDemo` example discussed in section 4.1 and `perform()` method from the `CreateUserHTMLAction`

```
1  //GUI event-handling example
2     public void actionPerformed(ActionEvent e) {
3          if (e.getActionCommand().equals("disable")) {
4              b2.setEnabled(false);
5              b1.setEnabled(false);
6              b3.setEnabled(true);
7          } else {...}
8    }
9  //J2EE event-handling example
10   public Event perform(HttpServletRequest request)
11     throws HTMLActionException {
12
13     String userName = (String)request.getParameter("j_username");
14     String password = (String)request.getParameter("j_password");
15     String password2 = (String)request.getParameter("j_password_2");
16     ...
17         if (userName != null && password != null)
18             return new CreateUserEvent(userName,password);
19         return null;
20     }
21 }
```

Figure 5.8: *GUI and J2EE event-handling method examples*

event handling class, taken from the Pet Store example described in section 4.2. While these two event-handling methods use different APIs, they perform a similar task of identifying the contents of parameter object, which encodes user inputs, e.g., the type of button clicked (`"disable"`), the text field filled out (`"j_username"`). Note that constant strings are used to denote the type of button, text field, etc. Such constants are easily identified using a simple scanning algorithm, which walks over event-handling methods (using domain-specific information about event-handling APIs) and looks for constant strings used as parameters to domain-specific APIs used to unwrap objects that encode user events. The discovered information can be used to populate user event objects with concrete data.

Figure 5.9 shows examples of populating `ActionEvent` object used in the `ButtonDemo` driver and populating `HttpServletRequest` object used in the Pet Store driver.

```
1 //GUI event population example
2 ActionEvent actionEvent = new ActionEvent( "disable");
3
4 //J2EE event population example
5 HttpServletRequestImpl createUserEvent=new HttpServletRequestImpl("createUser");
6 createUserEvent.setParameter("j_username", Abstraction.TOP_STRING);
7 createUserEvent.setParameter("j_password", Abstraction.TOP_STRING);
8 createUserEvent.setParameter("j_password_2", Abstraction.TOP_STRING);
```

Figure 5.9: *GUI and J2EE event population examples*

Type	Concrete	Abstract	Choice
Bool	false	Abstraction.TOP_BOOL	Verify.randomBool()
Int	0	Abstraction.TOP_INT	Verify.randomInt(n)
String	”top”	Abstraction.TOP_STRING	Verify.randomObject(”String”)
C ∈ E	null	C.TOP_OBJ	Verify.randomObject(”C”)
C ∈ U	null	n/a	Verify.randomObject(”C”)

Table 5.1: *Value Generation for JPF Framework*

5.5 Code Generation

Environment code generation can be separated into: action code generation, pattern code generation, and generation of the outer shell for drivers and stubs. For LTL pattern, the tableau method from [34] is used to construct a transition system emitted as Java code. For regular expressions, pattern code is generated from the abstract syntax trees for regular expressions by matching patterns in the tree and emitting Java code. Actions form the leaves of regular expression syntax trees, hence code generation for actions is simply a method call from the regular expression code generation pass.

5.5.1 Action Code Generation

As we described in section 5.3.1, users can describe program actions using various Java expressions including special modeling primitives used to denote *choice* and *abstract* values. Users may also omit specifying values. To fill omitted values, BEG performs types checking to calculate the appropriate types of values. There are three categories of values BEG can emit: concrete, abstract, and choice values. Table 5.1 shows value code generation for some

```
r|s          →  switch (chooseInt(1)) {
                  case 0: code(r); break;
                  case 1: code(s); break;
                }
r;s          →  code(r); code(s);
r*           →  while (chooseBool()) {code(r);}
r?           →  if (chooseBool()) {code(r);}
r+           →  do { code(r);} while (chooseBool())
r + {n}      →  for (int i=0; i<n; i++) { code(r);}
r + {n,m}    →  for (int i=0; i<n+chooseInt(m-n); i++) { code(r); }
```

Figure 5.10: *Assumption Semantics*

common types of values. By default, BEG emits `TOP` values for scalars, Strings and types belonging to the environment and choice values for reference types belonging to the unit.

For example, a query for method `m` with an empty formal parameter type list on class C described in Section 5.3.1 would return the full descriptions of methods 1, 2 and 3. Code would then be emitted that allowed for each of the possible calls as follows:

```
if (chooseBool()) {
  chooseClass("C").m(chooseClass("P"), chooseClass("Q"));
} else if (chooseBool()) {
  chooseClass("C").m(chooseClass("P"));
} else {
  chooseClass("C").m();
}
```

5.5.2 Pattern Code Generation

Regular expression assumption specifications are mapped to Java using the templates shown in Figure 5.10; for clarity we use r and s to refer to distinct instances of **spec** from the syntax. These templates use the non-deterministic choice constructs mentioned previously and are defined recursively, using *code* to refer to the code fragment for a given subexpression. For expressions that are program actions, the *code* call generates code as described in the preceding section.

Much of the behavior of generated model code is internal to the environment. Internal environment states and actions are *hidden* in our models by embedding them in atomic statements. Atomic statements are defined by pairs of `beginAtomic()` and `endAtomic()`

method calls. No lexical structuring of these calls is required, rather an atomic statement extends along any path from an instance of `beginAtomic()` to an instance of `endAtomic()`. We hide environment details by emitting `endAtomic()` calls immediately before the code for a program action and `beginAtomic()` calls immediately after the code for a program action. Additionally, the first statement in each environment thread is `beginAtomic()` and the last is `endAtomic()`. This strategy has two consequences: internal environment behavior does not contribute to state explosion and internal actions are elided from counter-examples making them shorter and easier to read.

5.5.3 Driver and Stub Code Generation

Throughout this thesis, we give many examples of generated drivers and stubs. Section 3.1 presents a driver generated from regular expressions and stubs generated from side-effects summaries for the observer-observable example, section 7.1 presents a driver generated from regular expressions and stubs for NASA's autopilot tutor example, section 7.2 presents stubs generated for GUI libraries, section 7.3 presents drivers generated from regular expressions and stubs for two modules of Fujitsu's I-BPM software, and section 7.4 presents a driver generated from regular expressions and stubs for J2EE components. In addition, Appendix A gives more examples, e.g, Appendix A.1.1 shows code for the universal driver for the observer-observable example and Appendix A.1.2 shows observer's stubs generated from user specifications.

5.6 Limitations

While the BEG specification language is rich enough to support most of Java constructs, it does not have support for all of Java. Some Java statements can be specified using the current support, e.g., the `if` statement can be specified using the `?` operator, loops can be specified using the `*` operator. However, some features can not be simulated, for example, there is no support for the `synchronized` blocks inside method bodies. Many

synchronization aspects of the environment can be encoded: in the driver specifications, one can specify the driver threads and in stub specifications, one can include `synchronized` into the environment methods' signature. However, there is not current support for inner synchronization, e.g., one cannot specify an environment method that takes its parameter and modifies it, while holding a lock on it. None of our case studies needed support for such a feature. However, in future, the BEG specification language can be extended to handle more of Java expressions and statements, including support for a method's inner `synchronized` blocks and property specification, e.g., assertions.

Similarly, static analysis in BEG does not calculate all possible unit-environment interactions. There are techniques such as slicing, which can calculate all possible unit-environment dependencies. However, slicing usually requires closed pure Java programs, is more expensive than environment generation, and produces fewer reductions. By design, for scalability, BEG skips some dependencies (e.g., concurrency-related dependencies). This may be appropriate, since most stubs execute atomically, with no interference [28]. Static analyses in BEG were driven by case studies, which showed that data effects were effective for mining domain-specific features like containment.

Chapter 6

BEG Implementation and Usage

BEG is implemented in Java on top of the Soot framework [91, 76]. BEG uses the following Soot features: at the front end, BEG uses Soot's class loading; to perform static analysis, BEG uses Soot's class hierarchy, method control flow graph and call graph representations; to perform interprocedural side-effects analysis, BEG uses Soot's intraprocedural DFA framework. We discuss the usage of each Soot feature in the following sections.

6.1 High-Level Architecture

Figure 6.1 shows a high-level architecture of BEG. There are four main modules in BEG: interface finder, assumptions acquirer, code generator, and code printer. BEG has an extensible plug-in architecture. Each of the modules includes an abstract class that can be extended and customized for specific domains or user needs. `ApplInfo` data structure is used to carry information about the application under analysis from one module in BEG to another. `ApplInfo` is also extensible and can encode domain-specific information about the application under analysis. Each BEG module adds information to the `ApplInfo` instance being passed.

The main class, `EnvGenerator`, reads a configuration file and creates instances of the following types, according to the configuration file: `ApplInfo`, `InterfaceFinder`, `AssumptionsAcquirer`, `CodeGenerator`, and `CodePrinter`. The main class also loads unit classes using Soot, which loads and stores class files as instances of `SootClass`. A `SootClass` has information about

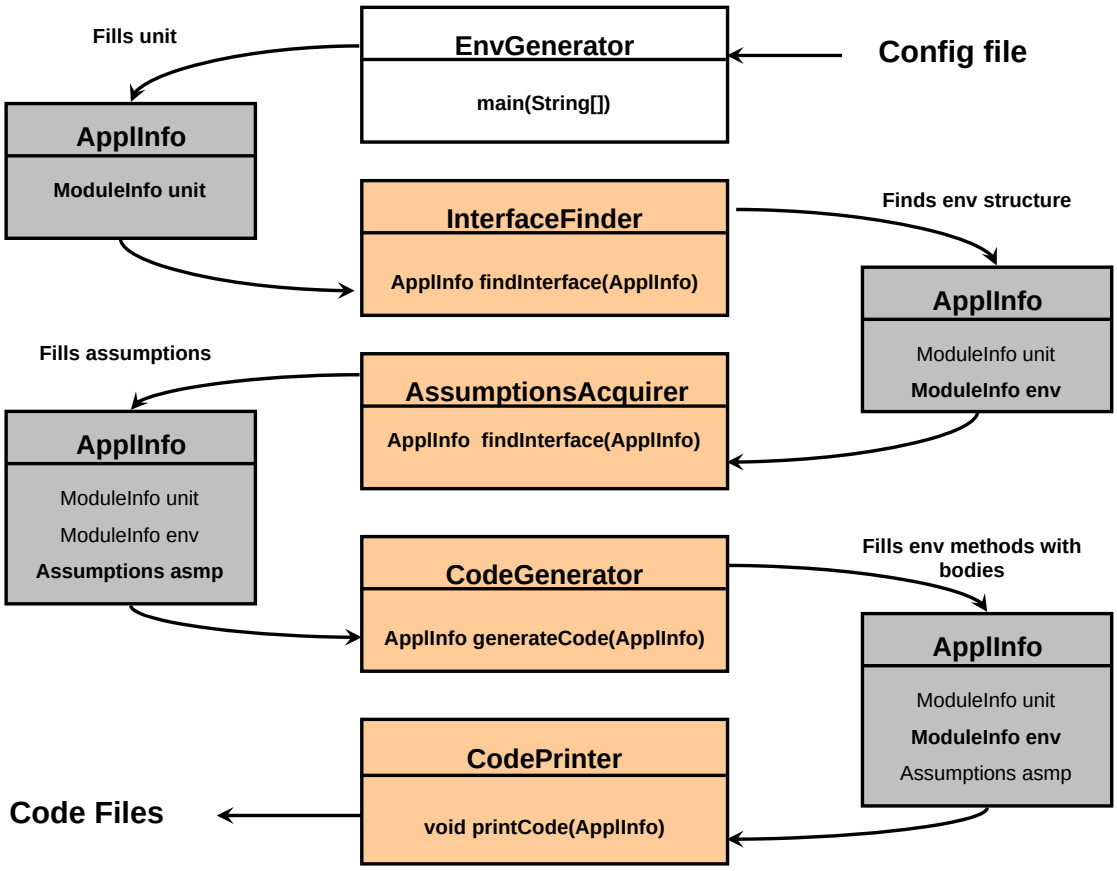

Figure 6.1: *BEG High Level Architecture*

the name of the class, its parent, the interfaces it implements, a list of `SootMethods` and `SootFields` of the class. A `SootMethod` has information about the name of the method, its modifiers, its parameter types, return type, and a body, which consists of a list of locals and a list of statements. Soot has support for several intermediate representations to encode method body information: Baf, Jimple, and Grimp (see Soot tutorials [75]). In this thesis, we use Jimple, a three-address bytecode representation which offers typed variables and a limited number of statement kinds. Jimple is a convenient representation for data flow

```
1  public abstract class ApplInfo {
2    ModuleInfo unit;
3    ModuleInfo env;
4    EnvHierarchy hierarchy;
5    EnvCallGraph callGraph;
6    Assumptions assumptions;
7
8    //constructors
9    //get methods
10   //set methods
11   //domain defining methods
12   abstract boolean isRelevantType(Type type);
13   abstract boolean isRelevantClass(SootClass sc);
14   abstract boolean isRelevantMethod(SootMethod sm);
15   abstract boolean isRelevantField(SootField sf);
16 }
```

Figure 6.2: *ApplInfo Class*

analyses.

Next, we describe each module in greater detail. All BEG classes belong to `edu.ksu.cis.envgen.*` packages. For brevity, we drop `edu.ksu.cis.envgen` from the package names.

6.1.1 Application Information

The `ApplInfo` data structure is used to carry information through the pipeline of BEG modules. Figure 6.2 shows excerpts of its implementation. `ApplInfo` declares the following fields used to store information about the program under analysis:

- `ModuleInfo unit`: This field keeps track of the unit classes, loaded by BEG's main class.
- `ModuleInfo env`: This field keeps track of the environment classes, methods, and fields discovered by the `InterfaceFinder` module.
- `EnvHierarchy hierarchy`: This field keeps information about the hierarchy of the program under analysis. `EnvHierarchy` may be different from the Soot's hierarchy because BEG-generated environment is usually smaller than the actual environment.

- `EnvCallGraph callGraph`: This field stores information about the call graph of the program under analysis. `EnvCallGraph` may different from the Soot's call graph because BEG can prune Soot's call graph based on domain-specific information.

- `Assumptions assumptions`: This field gets filled by the `AssumptionsAcquirer` and is used by the `CodeGenerator` to build bodies for environment methods.

The `ApplInfo` is an abstract class, which declares `isRelevant*()` methods used to describe domain-specific information. These methods are used by various BEG modules:

- `isRelevantType(Type type)`: describes relevant types. This method is used by the points-to analysis to track points-to information for objects of relevant types only, e.g., unit type.

- `isRelevantClass(SootClass sc)`: describes relevant classes. This method is used by the unit interface finder to produce a smaller set of entry points into the unit, e.g., for GUI domains, only event-handling classes need to be considered.

- `isRelevantMethod(SootMethod sm)`: describes relevant methods. This method is used by the unit interface finder to produce a smaller set of entry points into the unit, e.g., for GUI domains, only event-handling methods need to be considered. In addition, information about irrelevant methods can be used to prune Soot's call graph.

- `isRelevantField(SootField sf)`: describes relevant fields. This method is used by the side-effects analysis to track side-effects to specific fields only.

BEG includes several implementations of the `ApplInfo`:

- `DefaultDriverInfo`: Considers all public methods and fields as relevant. This class can be used to generate universal or domain-unspecific drivers.

- `DefaultStubInfo`: Describes unit type fields as relevant. This class can be used to generate stubs with side-effects to objects of unit type.

- `GUIDriverInfo`: Considers special event-handling APIs from Swing/AWT libraries as relevant. This class can be used to generate drivers for GUI applications.
- `GUIStubInfo`: Encodes information about specific fields in Swing/AWT libraries, as discussed in section 4.1.
- `J2EEDriverInfo`: Considers special event-handling APIs from J2EE library as relevant. This class can be used to generate drivers for J2EE applications.
- `J2EEStubInfo`: Encodes information about specific fields in J2EE library, as discussed in section 4.2.

Next, we describe BEG modules and how they use the `ApplInfo` data structure in greater detail.

6.1.2 Interface Finders

The `InterfaceFinder` class is an abstract class with the following methods:

```
public abstract class InterfaceFinder {
  public abstract void setOptions(Properties properties);
  public abstract ApplInfo findInterface(ApplInfo applInfo);
}
```

There are two aspects of the unit-environment interface as discussed in section 5.2 and there are two implementations of `InterfaceFinder` in BEG:

- `UnitInterfaceFinder`: implements the algorithm described in section 5.2.1: it walks over unit classes and gathers relevant methods and fields. The algorithm uses implementation of `isRelevantClass()`, `isRelevantMethod()`, and `isRelevantField()` to gather domain-specific entry points into the unit under analysis. This information is later used for driver generation.

  One can extend `UnitInterfaceFinder` to collect specific classes, methods, and fields by providing their own implementation of `isRelevant*()` methods.

- `EnvInterfaceFinder`: implements the algorithm described in section 5.2.2: it walks over unit classes and finds all external references to classes, methods, and fields. For each external reference, `EnvInterfaceFinder` creates a new `SootClass`, `SootMethod`, or `SootField` and stores them in the `ModuleInfo env` field of the `ApplInfo` object. Note that `EnvInterfaceFinder` discovers the structural information about the environment, not its behavior.

  Before scanning the unit, the `EnvInterfaceFinder` builds a call graph. This is done to resolve virtual invoke expressions. There are two options available in BEG: one based on Class Hierarchy Analysis (CHA) and one based on call graph build by Spark [53]. The second one is done using a whole-program analysis and requires presence of the main class and main method. Thus, the Spark call graph can be used only after driver generation phase. CHA-based call graph can be used without a main class, however, it is less precise.

6.1.3 Assumptions Acquirers

BEG has support for acquiring assumptions from two sources: user specifications and static analysis.

User Specification

The module for reading user specifications includes a JavaCC-generated parser, based on the specification language grammar described in section 5.3. The `spec.SpecReader` class extends `AssumptionsAcquirer`, parses, type checks a user specification and produces `Assumptions` object which encodes the specification information. The `Assumptions` object is later used by driver or stub generators to produce bodies for environment methods.

Static Analysis

Currently, BEG includes `analysis.SideEffectsAnalyzer`, which extends `AssumptionsAcquirer` and walks over external references discovered by `EnvInterfaceFinder`, performing the inter-

procedural, compositional, parameterized, flow-sensitive points-to and side-effects analysis described in section 5.4.

BEG uses Soot's DFA framework, which, given an implementation of transfer functions, merge operator, direction of data flow, and initial value, performs a fixed point computation over the control flow graph of each method. Soot's DFA framework is interprocedural, i.e., it walks over CFG for each method. We extend it by implementing transfer functions for invoke statements, i.e., by building the interprocedural CFG on the fly.

BEG uses domain-specific information, encoded in `isRelevant*()` methods of the `ApplInfo` instance to (1) scope the call graph and interprocedural CFG based on `isRelevantMethod()` (2) scope points-to analysis based on `isRelevantType()`, and (3) scope side-effects analysis based on `isRelevantField()`. Depending on the implementation of `isRelevant*()` methods, BEG can be tuned to produce side-effects to all unit-type fields, fields specific to Swing/AWT libraries, or fields specific to J2EE libraries. One can provide their own implementation of `isRelevant*()` methods to collect side-effects to specific objects in the environment.

6.1.4 Code Generators

A code generator uses information encoded in the `Assumptions` object to build bodies for environment methods discovered by the interface finder. As mentioned, we use Jimple to perform scanning and static analysis techniques. We extended the Soot framework with classes that represent Java bodies, Java statements and Java expressions. Code generators build Java bodies and attach them to `SootMethod`s.

Driver Generators

Driver generators build bodies for driver class methods. The following driver generators are implemented in BEG:

- `UnivDriverGenerator`: builds run methods for universal threads, which perform all possible sequences of actions on the unit.

- `SpecDriverGenerator`: builds run methods based on user specifications, as described in section 5.5.

- `J2EESpecDriverGenerator`: builds a setup section according to mappings from events to event handlers and, for each event, stamps out event-handling code customized according to J2EE APIs, as described in section 4.2 and applied in section 7.4.

Stub Generators

BEG has implementations of the following stub generators:

- `EmptyStubGenerator`: builds empty bodies, inserts `return` statements if necessary.

- `SpecStubGenerator`: builds method bodies based on user specifications, as described in section 5.5.

- `SEStubGenerator`: builds method bodies based on side-effects summaries produces by side-effects analysis.

6.1.5 Code Printers

The `CodePrinter` is an abstract class that declares methods that can be configured to produce code for different languages or frameworks:

```
1  public abstract CodePrinter{
2    public abstract void setOptions(Properties properties);
3    //class structure
4    public abstract void printClass(SootClass sc, FileWriter file);
5    public abstract void printMethod(SootMethod sm, FileWriter file);
6    ...
7    //modeling primitives
8    public abstract void public String printTopValue(Type type);
9    public abstract void String printRandomObjectCall(Type type);
10   ...
11 }
```

Currently, BEG provides implementation for `JavaPrinter`, which produces Java code. One can extend the `JavaPrinter` to produce modeling primitives for different model check-

ing frameworks. By default, the JavaPrinter produces modeling primitives supported in JPF.

Overall, BEG contains about 20K LOC, with 8K taken by JavaCC-generated files, which implement a parser for BEG specification language.

6.2 BEG Options

BEG has a command line interface. The following command is used to run BEG:

```
java edu.ksu.cis.envgen.EnvGenerator -c <configfile>
```

where `<configfile>` is a name of the configuration file, which specifies concrete classes to instantiate for each of the BEG modules, various BEG options, and unit classes.

Table 6.1 lists the BEG options (second column) and their values (third column). Columns D (fourth column) and S (fifth column) denote whether the option is used to generate drivers or stubs. Some options can be used for both. Default values are shown in bold font.

Options 1-5 show BEG's extensible classes and their implementations available in BEG. We previously described each of the implementations.

Options 6-16 can be used for driver generation and options 7-21 can be used for stub generation. Most options are self-explanatory. Options `model` and `ignoreModeling` can be used to scope code generation; options `analyze` and `ignoreAnalyzing` can be used to scope static analysis, in addition to implementations of `isRelevant*()` methods; option `unitAnalysis` can be used to force BEG to analyze unit classes. This option can be used to analyze library packages without client code: by loading library packages as unit and setting `unitAnalysis` to true.

6.3 Common Configurations

In this thesis, we employ user specifications to generate drivers and static analysis to generate stubs. Figure 6.3 shows the flow of these two approaches. Next we describe several configurations used for the case studies in this thesis.

	Option	Value	D	S	Description
01	DomainInfo	**DefaultDriverInfo**	X		unit public methods and fields
		GUIDriverInfo	X		GUI event handling APIs
		J2EEDriverInfo	X		J2EE event handling APIs
		DefaultStubInfo		X	unit types
		GUIStubInfo		X	GUI stub features
		J2EEStubInfo		X	J2EE stub features
02	InterfaceFinder	**UnitInterfaceFinder**	X		finds unit interface
		EnvInterfaceFinder		X	finds all external references
03	AssumptionsAcquirer	SpecReader	X	X	reads a spec file
		SideEffectsAnalyzer		X	performs side-effects analysis
04	CodeGenerator	**UnivDriverGenerator**	X		universal driver
		SpecDriverGenerator	X		driver based on specs
		EmptyStubGenerator		X	empty stubs
		SpecStubGenerator		X	stubs based on specs
		SEStubGenerator		X	stubs based on se summarries
05	CodePrinter	**JavaPrinter**	X	X	prints Java code
06	numThreads	<int> (default **2**)	X		number of driver threads
07	specFileName	<fileName>	X	X	specification file name
08	printActions	true/**false**	X	X	flag to print env actions
09	outputDir	<dirName>	X	X	output directory
10	outputPackage	<packageName>	X	X	package prefix
11	outputValue	**concrete**/choice/abstract	X	X	output values
12	framework	**jpf**/bogor	X	X	modeling primitives support
13	unit	<classNames>	X	X	unit classes
14	unitPath	<pathNames>	X	X	unit directories and jars
15	model	<pathNames>	X	X	model only these packages
16	ignoreModeling	<pathNames>	X	X	do not model these packages
17	analyze	<pathNames>		X	analyze only these packages
18	ignoreAnalyzing	<pathNames>		X	do not analyze these packages
19	mainClass	<className>		X	main class
20	callGraph	**cha**/spark		X	type of call graph
21	unitAnalysis	true/**false**		X	flag to analyze unit classes

Table 6.1: *BEG Options*

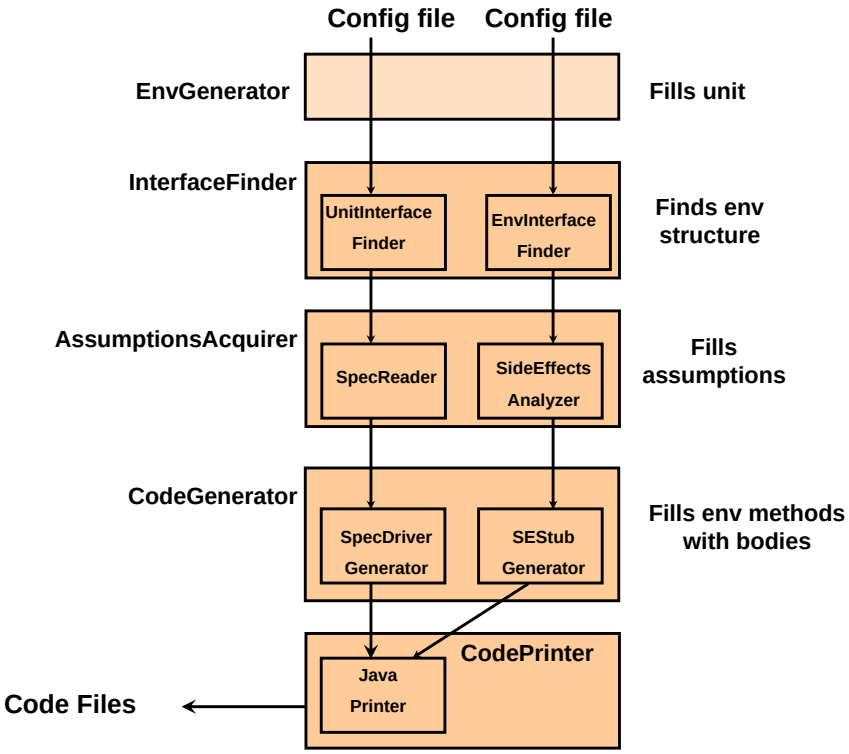

Figure 6.3: *BEG Common Configurations for Driver and Stub Generation*

6.3.1 Driver Generation

- **Universal Driver Generation**

  Appendix A.1.1 shows configuration file for generation of universal drivers for the observer-observable example, described in section 3.1.

- **Using Specifications**

  Configuration for driver generation for the observer-observable example, described in section 3.1.

```
ApplInfo = applinfo.domain.DefaultDriverInfo
InterfaceFinder = applinfo.UnitInterfaceFinder
AssumptionsAcquirer = spec.SpecReader
```

```
CodeGenerator = codegen.SpecDriverGenerator
specFileName = specs/observer-re.spec
unit = Subject Watcher
```

- **Domain-Specific Driver Generation**

  Configuration used to generate drivers for the Pet Store example described in section 7.4.

```
ApplInfo = applinfo.domain.J2EEDriverInfo
InterfaceFinder = applinfo.UnitInterfaceFinder
AssumptionsAcquirer = spec.SpecReader
CodeGenerator = codegen.SpecDriverGenerator
specFileName = specs/petstore.spec
unit = petstore classes
```

6.3.2 Stub Generation

- **Empty Stubs** Appendix A.1.3 shows the BEG configuration for generation of empty stubs for the observer-observable.

- **User Specifications** Appendix A.1.2 shows the BEG configuration for generation of stubs for the observer-observable from user specifications.

- **Side-Effects Analysis** Configuration for stub generation for the observer-observable example, using side-effects analysis:

```
ApplInfo = applinfo.domain.DefaultStubInfo
InterfaceFinder = applinfo.EnvInterfaceFinder
AssumptionsAcquirer = analysis.data.SideEffectsAnalyzer
CodeGenerator = codegen.SEStubGenerator
unit = Subject Watcher
```

- **GUI Components Analysis**

  The following configuration is used to process the button demo example described in section 4.1:

```
ApplInfo = applinfo.domain.GUIStubInfo
InterfaceFinder = applinfo.EnvInterfaceFinder
AssumptionsAcquirer = analysis.data.SideEffectsAnalyzer
CodeGenerator = codegen.SEStubGenerator
unit = ButtonDemo
```

- **J2EE Components Analysis**

  The following configuration is used to process the Pet Store example, described in section 7.4:

```
ApplInfo =  applinfo.domain.J2EEStubInfo
InterfaceFinder = applinfo.EnvInterfaceFinder
AssumptionsAcquirer = analysis.data.SideEffectsAnalyzer
CodeGenerator = codegen.SEStubGenerator
unit-path = <petstore path>
```

- **Libraries without Client Code**

  BEG can be configured to analyze libraries without supplying their client code.

```
unitAnalysis = true
unitPath = <lib jars>
```

  When the flag `unitAnalysis` is on, BEG will load unit classes and analyze them. This option is useful when stubs are needed for a library package without knowing which specific classes are used ahead of time.

6.4 Limitations

BEG is configured to run once, to generate drivers or stubs but not both. The tool can be extended to generate both drivers and stubs on the same run. BEG can also be extended to run multiple times and be invoked from another tool.

Chapter 7

Experience

We have applied BEG to a variety of examples. A number of multi-threaded Java programs that have been the subject of analysis in literature have been re-verified by generating the previously hand-built environments with BEG. In addition to the Observer-Observable example, these examples include: a Producers-Consumers framework for exercising a bounded buffer, a generic Readers-Writers synchronization framework, and dining philosophers with host, a classic synchronization problem.

While BEG proved to be useful in generating environments for these small systems, the tool support is much more valuable when attempting to reason about properties of larger software systems. Our first large case study was NASA's Autopilot simulator [74], described in section 7.1. This application is implemented as Java applet with a GUI implemented using Swing and AWT libraries. The Autopilot motivated our domain-specific approach to environment generation for GUI applications, discussed in section 4.1. Section 7.2 presents our experience with model checking a collection of GUI examples, including the `ButtonDemo`, described in 4.1.

Next, we verified two modules belonging to Fujitsu Enterprise software, which were previously verified with manually constructed environments. We were able to re-generate the environments without much domain knowledge and compare performance of BEG-generated environments with the manually built ones. We describe this experiment in section 7.3. After that, a research group at Fujitsu, Japan, asked us to set up verification of J2EE

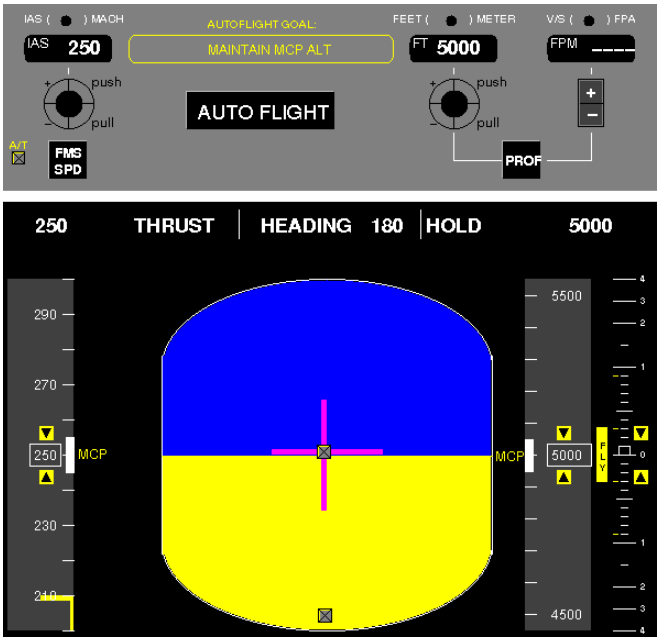

Figure 7.1: *Autopilot Tutor GUI*

applications using JPF. They had an internal framework for developing web applications and wanted to evaluate model checking. As a proof of concept, they picked SUN's Pet Store example [82], whose architecture and event-handling mechanism was similar to their internal applications. Thus, we embarked on verification of a whole J2EE application, not just its separate modules. We studied J2EE frameworks and customized BEG to handle J2EE-specific APIs, as described in section 4.2. Section 7.4 describes our experience with the Pet Store example, which was introduced in section 4.2.

7.1 NASA's Autopilot Tutor

In this section we describe environment generation and model checking results for an MD-11 Autopilot tutor [74]. This section is based on results previously reported in [86, 88].

The Autopilot tutor is a web-based application with a GUI that simulates the Autopilot Mode Control Panel (MCP) and a Primary Flight Display (PFD) of an MD-11 aircraft autopilot. Figure 7.1 shows the GUI of the tutor. A user may click on the buttons to dial desired altitude and vertical speed, and advance the aircraft towards its goal altitude. The autopilot tutor is implemented as an applet. The application code consists of more than 3500 lines of code clustered in one class. These measures bely the true complexity of the system as there is intensive use of `java.awt` and `java.swing` GUI frameworks that influences the behavior of the system; in fact the main thread of control is owned by the framework and application methods are invoked as application call-backs.

The autopilot tutor was checked for *automation surprises*, a term used to describe scenarios in Human-Machine Interaction (HCI) when a machine behaves differently from the user's expectation. An example of an automation surprise is the well known *mode confusion* problem in civil aviation, where a pilot thinks that the autopilot is operating in one mode, when in fact, it is operating in another. Specifically, in the case of the *kill the capture* mode confusion [62], a pilot thinks he is going to capture his target altitude but the autopilot misses the altitude.

Among the underlying causes of automation surprises are the complexity of automation behavior (there may be many modes of operation, some unaccounted for), the user's lack of understanding of the machine behavior (inadequate user model), and the inability of the interface to provide sufficient information to the user to unambiguously determine the state of the machine (inadequate interface). All of these might lead to a situation where the *pilot's mental model* of the system behavior, i.e., the pilot's understanding of how the system should behave, is different from the actual system behavior.

To verify the autopilot tutor, one needs to extract models of the following components

in the system: (1) the user model (the pilot's mental model), (2) the task the user is trying to achieve (e.g., "take the aircraft to a certain altitude"), (3) the machine (the autopilot), and (4) the interface between the user and the machine (knobs, wheels, and displays in a cockpit) [18]. Next, we use our environment generation technique for constructing these models and verifying their interaction. Specifically, the main class of the application, the `Autopilot`, which extends `java.applet.Applet` and makes a large number of calls to `AWT` methods to create and update the simulated cockpit displays, is treated as a unit under analysis. The GUI components are stubbed out. The user tasks are used to construct drivers and a pilot's mental model is used to encode and check assertions in the code during verification.

7.1.1 Driver Generation

Next, we describe some of the pilot's tasks and sequences of actions that correspond to achieving such tasks. All pilot's tasks and their corresponding scenarios were taken from the autopilot tutorial [74].

Pilot's Actions

All of the tasks described in this section are achieved using combinations of clicking on the following buttons on the display:

- the *mcp altitude knob*, which has four clickable areas: two symbols, plus and minus, used to increment, and respectively decrement, the mcp altitude in increments of 100 (we will call these `inrcMCPAlt` and `decrMCPAlt` actions); and two symbols, pull and push, representing pulling and pushing the knob (we will call theses `pullAltKnob` and `pushAltKnob` actions).

- the *mcp pitch wheel*, which has plus and minus symbols for incrementing and decrementing of mcp vertical speed in increments of 100 (we will call these `incrMCPVS/decrMCPVS` actions).

```
1  public void mouseClicked (MouseEvent e){
2    //MCP Alt Knob PULL
3    if ((e.getX()> mcpx+altcntx) && (e.getX() < mcpx+altcntx+(45/2)) &&
4       (e.getY()> mcpy+knobsy) && (e.getY() < mcpy+knobsy+(45/2)))
5    {
6      MCPKnobWheeln = 1;
7      mcpAltitude = mcpPreSelAltitude;
8    }
9
10   //MCP Alt Knob PUSH
11   if ((e.getX()> mcpx+altcntx) && (e.getX() < mcpx+altcntx+(45/2)) &&
12   (e.getY()> mcpy+knobsy-(45/2) ) && (e.getY() < mcpy+knobsy  ))
13   { MCPKnobWheeln = 2;
14     if (vs >= 0) {mcpAltitude=((altitude+captureRegion)/100)*100;}
15     else{mcpAltitude=((altitude-captureRegion)/100)*100;}
16     mcpPreSelAltitude=mcpAltitude;
17   }
18   ...
19 }
```

Figure 7.2: *Snippet of the mouseClicked Method*

- the *fly button*, which is used to step the aircraft forward (we will call it the `fly` action).
- the *init button*, which resets the altitude of the aircraft to 5000 (we will call it the `start` action).

As discussed in section 4.1.2, each time a user clicks on a button or performs any other action on a screen of a GUI, an object of type `Event` is fired and sent to objects of type `Listener`. Listener objects examine the event object and invoke a special event handling code that corresponds to that particular type of event. This may result in the state change of the underlying application as well as the state of the GUI itself.

In the case of the autopilot tutor, the display is a `MouseListener` on itself, and it implements the `mouseClicked(MouseEvent e)` method, which, depending on where on the screen the event was originated, invokes a different piece of code. Figure 7.2 shows a snippet of code from this method. To detect the type of the event, the event-handling code inspects the coordinates of the event objects. For each event, we manually inspected its coordinates, for example, to simulate the `incrMCPAlt` action, we need to create an event with $X =$

$400, Y = 110$ and pass it to the `mouseClicked` method.

Pilot's Tasks

Next, we describe some scenarios described in the autopilot tutorial:

- **Climb/Descend and Maintain MCP Altitude**

  This goal is achieved by dialing the MCP altitude, pulling the altitude knob, and flying the aircraft forward until the desired altitude is reached. More formally, we need to execute the `inrcMCPAlt/decrMCPAlt` action some number of times, followed by the `pullAltKnob` action, followed by the `fly` action as many times as required to meet the goal.

  Using the regular expressions notation, we can describe this scenario using the following expression:

  ```
  incrMCPAlt/decrMCPAlt*; pullAltKnob; fly* // until level off
  ```

- **Capture MCP Altitude**

  The aircraft automatically transitions to this goal as it approaches the desired altitude to level flight. This goal can be achieved by dialing the mcp altitude, pulling the mcp altitude knob and stepping the aircraft forward until it is in a capture region. The formal representation can be described as follows:

  ```
  incrMCPAlt/decrMCPAlt*; pullAltKnob; fly* // until in capture region
  ```

- **Climb/Descend and Maintain MCP - Fixed Rate of Climb/Descend (ROC/ROD)**

  This mode is achieved by rotating the pitch wheel (setting the vertical speed) while the aircraft's goal is Climb/Descend Maintain MCP Altitude.

  ```
  incrMCPAlt/decrMCPAlt*; pullAltKnob; incrMCPVS/decrMCPVS*; fly*
  ```

- **Climb/Descend Away from MCP Altitude - 2 sec**

```
environment{
  definitions{
    pullAltKnob=mouseClicked(pullAltKnobEvent);
    incrMCPAlt=mouseClicked(incrMCPAltEvent);
    incrMCPVS=mouseClicked(incrMCPVSEvent);
    fly=mouseClicked(flyEvent);
  }
  driver assumptions{
    re{
      Main:
        Event incrMCPAltEvent = new MouseEventImpl(400, 110);
        Event incrMCPVSEvent = new MouseEventImpl(540, 110);
        Event pullAltKnobEvent = new MouseEventImpl(420, 130);
        Event flyEvent = new MouseEventImpl(550, 440);
        Autopilot autopilot = new Autopilot(); #

      User: init(); incrMCPAlt*; pullAltKnob; fly*; incrMCPVS*; fly* #
    }
  }
}
```

Figure 7.3: *Autopilot Assumptions*

This goal is achieved by rotating the pitch wheel while the aircraft is in the capture region and the vertical speed is low enough to allow the aircraft to stay in the capture region after 2 sec and re-enter the Climb/Descend MCP Altitude-Cap mode.

```
incrMCPAlt/decrMCPAlt*; pullAltKnob;  fly*; //until in capture region
incrMCPVS/decrMCPVS*; //small enough to stay within capture region
fly*
```

- **Climb/Descend Away from MCP Altitude**

  This goal is achieved by rotating the pitch wheel while the aircraft is in the capture region and the vertical speed is too high for the aircraft to stay within the capture region.

```
setMCPAlt*; pullAltKnob; fly*; //until in capture region
setVerticalSpeed*; // too large to stay within capture region
fly*
```

```
1 public class MouseEventImpl implements MouseEvent
2 {
3   int X;
4   int Y;
5   public MouseEventImpl(int p1, int p2){
6     X = p1;
7     Y = p2;
8   }
9 }
```

Figure 7.4: *MouseEvent Stub*

Driver Assumptions

For this system, we found it useful to name the user actions to improve the readability of both the assumption specifications and generated counter-examples. As shown in Figure 7.3, we used BEG's facility for defining mnemonics for driver actions. The assumptions were defined in terms of those mnemonics.

7.1.2 Stub Generation

BEG calculated the data effects of the `AWT` methods called from the `Autopilot` class and generated safe approximation of the data effects on explicitly defined fields of `Autopilot` and on fields inherited from `AWT` classes.

7.1.3 Verification Results

The autopilot model was checked for mode confusion problems by encoding a model of a user's understanding of the aircraft state. That user model was integrated with the system to monitor the state of the autopilot. Assertions were inserted to compare the state of the autopilot to the state of the pilot's mental model; assertion violations indicated a mismatch between the user model and the software's state, which implies a potential mode confusion. Next, we describe a simple pilot's model designed to catch "kill the capture" errors.

```
1  import gov.nasa.jpf.jvm.Verify;
2  public class PilotMentalModel{
3    private Autopilot ap;
4
5    public static final byte climb = 1;
6    public static final byte descend = -1;
7    public static final byte hold = 0;
8
9    public static byte expectation = hold;
10
11   public PilotMentalModel(Autopilot a){
12     ap = a;
13   }
14   public void getExpectation(){
15     if(ap.mcpAltitude - ap.altitude >= 100)
16       expectation = climb;
17     if(ap.altitude - ap.mcpAltitude >= 100)
18       expectation = descend;
19     if(ap.altitude == ap.mcpAltitude)
20       expectation = hold;
21     checkExpectation();
22   }
23   public void checkExpectation(){
24     Verify.assert(expectation != climb || ap.getMode() == climb);
25     Verify.assert(expectation != descend || ap.getMode() == descend);
26     Verify.assert(expectation != hold || ap.getMode() == hold);
27   }
28 }
```

Figure 7.5: *Pilot's Mental Model for Detecting Altitude Deviation Errors*

Pilot's Mental Model

We present a simple user model specifically built for identifying altitude deviation errors. The task description identifies specification classes for the states of the autopilot. We need to identify the states for the pilot model. Suppose, the user needs to identify whether the aircraft is climbing, descending, or holding the altitude. This suggests three specification classes: *climb*, *descend*, and *hold*. Figure 7.5 shows implementation of the `PilotMentalModel` class. The user model contains three states (modes) and each of the machine states is mapped to one of these categories, using method `getMode()`. The user mode is identified by the method `getExpectation()`: the user looks at the interface of the system and makes a prediction of whether the aircraft is climbing, descending, or holding

the altitude, e.g., if the goal altitude (`ap.mcpAltitude`) is greater than the current altitude (`ap.altitude`), then the aircraft is climbing. The method `checkExpectation()` checks that the predicted state of the machine corresponds to the appropriate actual state of the machine. We used JPF's assertions to encode the property (expected state $\Longrightarrow$ actual state), which is equivalent to (!expected state || actual state), as shown on lines 24-26.

Note that since "kill the capture" mode confusions occur when the aircraft misses its target altitude, i.e., keeps climbing or descending past the dialed altitude, having three modes in the pilot model allows us to catch such scenarios. For other mode confusion problems, we need to enrich the pilot's model to include more modes.

Model Checking Results

We refined driver assumptions, shown in Figure 7.3, with a call to `checkExpectation()` and bounded the number of actions by 10, as follows:

```
init(); incrMCPAlt^{1,10}; pullAltKnob;
(checkExpectation(); fly)^{1,10}; incrMCPVS^{1,10};
(checkExpectation(); fly)^{1,10}
```

Model checking the `Autopilot` class using a driver generated from the above specification, JPF produced a counter-example. The original error trace was given in terms of all Java statements included in the trace. When generating the autopilot driver, we turned the printing of environment actions on. Then we wrote a script that processes a JPF counterexample, containing `System.out.println("EnvDriver:<action name>")` and extracts a sequence of driver actions only. After processing, the counterexample looks as follows:

```
init; incMCPALT; incMCPALT; pullAltKnob; fly; fly; incMCPVS; fly
```

This trace corresponds to the following sequence of actions: increment mcp altitude twice (set the target altitude to 5200 starting from 5000), fly twice (advance towards 5200), increment the vertical speed once, and fly once (steps the aircraft to 5300). If a user performs these steps on the autopilot tutorial, he will see the aircraft climb past its goal altitude of 5200 and climb to 5300. In this state the assertions get broken: the user sees that the goal

altitude is below the current altitude and thinks that the aircraft should be descending, yet the aircraft is in the climbing mode.

It is interesting to note, that a previous effort to build an environment for this application required several months of manual work and yielded an environment model that was inconsistent with the actual environment implementation. From relatively simple specifications, and running a side-effects analysis on the GUI components of the application, BEG generated an environment in less than 4 minutes that was consistent with the implementation, modulo the fidelity of assumption specifications.

7.2 GUI Examples

In this section, we present our experience model checking a collection of GUI applications under all possible *interaction orderings* enforced by a GUI. These applications, taken from Java Swing tutorial [83], while not as large as many real applications, contain a representative collection of Swing components. This study appeared in [26].

7.2.1 Driver Generation: Event-Handling

As described in section 4.1, the event-handling mechanism in Swing applications is an example of publish-subscribe pattern. When a user performs an action on a GUI component, an event of corresponding type is fired and sent to listeners subscribed to be notified of that event.

Since all event-handling APIs are known a-priori, we can construct a universal driver that works for any GUI application written using Swing/AWT libraries. Figure 7.6 shows code for such a driver. The `main` method executes an infinite loop. Inside the loop, the program executes three statements (lines 7,8,10). In the method `chooseTopWindow()`, a window is selected for interaction by prioritizing modal dialogs and choosing any top-level window, or reachable sub-window, if none exist. That window is analyzed to determine the visible and enabled components it contains and one of those is selected (line 8). The registered handlers

```
1  public static void main(String[] args) {
2    JComponent container;
3    //setup GUI
4    ...
5    //event-handling loop
6    while (true) {
7      window = chooseTopWindow();
8      container = (JComponent) randomReachable("env.javax.swing.JComponent",
9                           window, isVisible, isEnabled);
10     notifyListeners(container);
11   }
12 }
13 public static Window chooseTopWindow() {
14   Window window = null;
15   Vector modalDialogs = SwingUtilities.getModalDialogs();
16   if (!modalDialogs.isEmpty())
17     window = (Dialog) modalDialogs.lastElement();
18   if (window == null) {
19     Vector topWindows = SwingUtilities.getTopWindows();
20     window = (Window) randomReachable("env.java.awt.Window",
21                                       topWindows);
22   }
23   return window;
24 }
25 public static void notifyListeners(JComponent container) {
26   EventListener[] list = container.getListeners();
27   ...
28   EventListener listener;
29   for (int i = 0; i < list.length; i++) {
30     listener = list[i];
31     if (listener instanceof ActionListener)
32       ((ActionListener) listener).actionPerformed(new ActionEvent(container));
33     if (listener instanceof ItemListener)
34       ((ItemListener) listener).itemStateChanged(new ItemEvent(container));
35     ...
36   }
37 }
```

Figure 7.6: *Universal Driver for GUI applications (excerpts)*

for that component are then notified in turn, using the method `notifyListeners()` (line 10), which collects all listeners registered on the event and invokes their event-handling methods. As a result of executing the event-handling code, the state of the GUI may change, therefore, the collection of top-level windows and their visible enabled components may change. The key to this model is the ability to express nondeterministic choice over collections of heap allocated objects.

Note that the universal driver is written manually once and is reused across multiple GUI applications. The universal driver checks all possible sequences of user actions allowed as well as constrained by a GUI. For applications with a complex state transition diagram, the number of all possible sequences can be large and model checking under the universal driver may be intractable. We did not encounter the state space explosion problem with a collection of GUI examples described in this section. However, if the infinite loop in the universal driver leads to a state space explosion, the loop can be bounded, e.g., by a length of use case scenario, or bounded model checking can be used.

7.2.2 Stub Generation: Swing/AWT Components

In this section, we give examples of stubs generated using BEG's side-effects customized for analyzing Swing/AWT libraries.

As described in section 4.1.2, all Swing components inherit their properties from class `java.awt.Component`, which declares boolean fields `visible` and `enabled`. `java.awt.Container` is a sub-type of `Component`, which implements containment properties through the field `Component[] component`. Modality is implemented by a boolean field `modal` of `java.awt.Dialog`. `JComponent`, a descendant of `Container`, declares `EventListenerList listenerList`, where listeners of `*Listener` type (e.g., `MouseListener`, `ComponentListener`) may register using `add*Listener()` method. All Swing components inherit this listener mechanism.

In addition to inherited features, Swing components declare fields reflecting their specific features, e.g., tabs implemented by `JTabbedPane`, which declares `Vector pages` to keep

```
1  public class Container extends Component{
2    Component[] component = new Component[0];
3    public Component add(Component comp) {
4      addImpl(comp, null, -1); return comp;
5    }
6    protected void addImpl(Component comp,
7                           Object constraints, int index){
8      if (ncomponents == component.length) {
9        Component newcomponents[]=new Component[..];
10       component = newcomponents; ...
11     }
12     if (index == -1 || index == ncomponents) {
13       component[ncomponents++] = comp;
14     } else {
15       component[index] = comp; ncomponents++;
16     } ...
17   }
18 }
```

Figure 7.7: *Example Swing Method add*

track of added tabs and `SingleSelectionModel` to keep track of tab selection. Integer type fields can affect the number of widgets created for a component (e.g., `int optionType` in `JPOptionPane` defines how many buttons are displayed on the pane). Fortunately, such fields have a predefined and small set of values (e.g., `optionType = YES_NO_OPTION` produces a pane with two buttons: `yes` and `no`).

We use BEG's specialized side-effects analysis to calculates side-effects to specific fields of Swing/AWT components, i.e., boolean fields `visible`, `enabled`; fields that serve as containers (arrays, lists) and `eventListeners`. We illustrate this analysis on the `add()` method of `java.awt.Container` class. Figure 7.7 shows excerpts of its Java implementation. We are interested in the side-effects this method has on the fields, discussed above, that are related to the abstract state of GUI components. Figure 7.8(top) shows the output of BEG that encodes the results of side-effects analysis. BEG calculates that the method must side-effect the field `component` by assigning the parameter object to an element of the array. Unfortunately, the Java code is complicated by various checks on the method input and the state of the field `component`. If the array is too small, the new array is allocated, and

```
// must side-effects
this.component[TOP_INT] = param0;
// may side-effects
this.component = new Component[TOP_INT];
this.component[TOP_INT]=chooseObject("Component");
```

```
1  public class Container extends Component {
2    Component[] component;
3    int length = MAX_SIZE;
4    int size = 0;
5    public Component add(Component param0){
6      component[size] = param0;
7      size++;
8      return param0;
9    }
10 }
```

Figure 7.8: *Method add Analysis and Model*

the elements from the old array are copied to the new array. It is hard to design static analyses to accurately track such behavior, therefore, the analyses results may be imprecise. However, the model-writer can inspect the analysis results and decide whether to model the behavior that causes the imprecision. In this case, we do not wish to model the allocation of the new array since any such error would be detected by simply executing the application. Therefore, the final model, shown in Figure 7.8(bottom), reflects only the must side-effects of the method.

Another advantage of using analysis results to guide or to check the modeling process is the ability of the analysis to identify the methods that do not have any side-effects on the specified fields. A majority of methods in Swing only effect the look and feel of a GUI and thus have no side-effects on interaction order related data. Such methods can be safely modeled using empty stubs.

7.2.3 Verification Results

To perform model checking of GUI examples, we used Bogor [71]. Bogor's architecture is designed to ease customization of its module to exploit properties of an application domain to reduce the cost of model checking. Bogor was customized to efficiently check the models

Example		Measure	ALL	SSC
Button Demo		Trans	1920	2045
Objects:	50	States	1816	7
Choices:	3	Space (Mb)	40.2	39.6
Locations:	7563	Time (s)	4	0.8
Voting Dialogs		Trans	3114	4630
Objects:	120	States	2930	17
Choices:	4	Space (Mb)	45.5	44.5
Locations:	8269	Time (s)	10	1
Dialog Demo		Trans	88493	181512
Objects:	257	States	84439	1033
Choices:	14	Space (Mb)	74.3	47.6
Locations:	8689	Time (s)	512	38
Calculator		Trans	29016	35574
Objects:	362	States	27903	105
Choices:	24	Space (Mb)	66.4	48.6
Locations:	8789	Time (s)	183	20

Table 7.1: *Verification Data for GUI Examples*

of GUI applications generated by our environment generation techniques. More specifically, it leveraged the fact that the user must wait for the GUI to respond to one request before he can input another action.

In contrast to general multi-threaded applications where interleaving may occur at each state, in models of single dispatch-threaded GUIs, branching in the state-space occurs only when nondeterministic choice constructs are used to model user selections or abstraction of the underlying application. Thus, the number of states stored could be reduced to those at which branching may occur and still preserve all user interaction orderings in the model. The solution was to modify the state storage strategy in Bogor to only store states in which a choice expression is invoked. The intuition is that those are the earliest points where we can decide whether the choices cause different states. This strategy was called store-states-on-choose **(SSC)**.

Figure 7.1 presents the results of running the applications on an Opteron 1.8 GHz (32-bit mode) with maximum heap of 1 Gb using the Java 2 Platform. For each example, we

give the total number of objects allocated during system execution (nearly all of which are Swing component and container sub-types), the number of nondeterministic choices used to model user inputs, and the number of control locations in the combined model of the Swing library, GUI implementation and underlying application; due to abstraction of the underlying application the actual number of lines of code for an example can be many times larger than the number of locations.

In all runs, we used all of the reductions and memory-compression techniques available in Bogor **(ALL)** and compared that to the addition of the store-states-on-choose **(SSC)** strategy.

We note that the time to generate the models for these systems was negligible, except for the manual process of reading side-effects summaries for Swing methods and pruning them based on our understanding of their actual behavior. It took several days to fine tune our model of Swing based on the approximate starting model. Fortunately, that process happens only once and its cost can be amortized across the analysis of many Java Swing applications.

The data clearly show the benefit of customizing the analysis for single dispatch-threaded GUIs; reductions of more than an order of magnitude in run-time are achieved for the examples. We note that memory reduction is not as apparent since these examples are relatively small. We expect that as GUI implementations scale, especially in terms of the number of non-modal dialogs, significant memory reductions can be observed.

7.3 Fujitsu's I-BPM

In this section, we present verification of two modules belonging to a development version of Fujitsu enterprise software called Interstage Business Process Management (I-BPM). This case study appeared in [89].

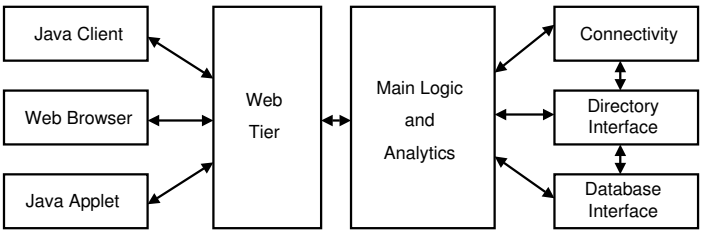

Figure 7.9: *I-BPM Architecture*

7.3.1 I-BPM Architecture

The I-BPM application has a 4-tier architecture, as shown in Figure 7.9. 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 [43]. The two modules that were analyzed were database adapter and cache, both residing in the third tier. All experiments were run on a desktop with 1G RAM, 2.4 GHz processor, running Linux 9.1, using SDK 1.5.

7.3.2 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 7.10. The client code:

1. acquires the `DbAdapter`

2. calls `getConnection()` on the `DbAdapter`, receiving a `DbConnection`

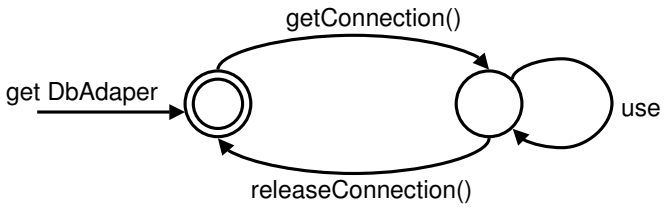

Figure 7.10: *Database Adapter Protocol*

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);
```

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 following code:

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

Automated stub generation presented in section 7.3.2 solves this problem by inserting nondeterminism into stubs automatically, without prior knowledge of the application.

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

Figure 7.11: *User Assumptions for Adapter Module*

Unit and Property Specification

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". All properties were specified using Java assertions and manually added to the driver code.

When generating environments with BEG, we started with a unit 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 unit later.

Driver Generation

Using the protocol in Figure 7.10 as a guide, we started with a specification shown in Figure 7.11. The specification file contains two sections: the setup section shows that the driver creates one instance of `DbAdapter` and the assumptions section shows that the driver spawns two threads. The methods used in the specification are the `DbAdapter` methods that implement the protocol in Figure 7.10. 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

```
1  public class EnvDriver {
2    public static void main(java.lang.String[] param0){
3    DbAdapterImpl adapter = new DbAdapterImpl("", 3);
4      try{
5          adapter.connect ("", "", "", "");
6
7      }catch(Exception e){e.printStackTrace();}
8      Client c1 = new Client(adapter).start();
9      Client c2 = new Client(adapter).start();
10   }
11 }
12 public class User extends java.lang.Thread {
13   public DbAdapterImpl adapter;
14   public Client(DbAdapterImpl param0){
15     adapter = param0;
16   }
17   public void run(){
18     try{
19       adapter.getConnection();
20
21       adapter.releaseConnection(
22       (DbConnection)Verify.randomObject("DbConnection");
23     }
24     catch(Exception e ){e.printStackTrace();}
25   }
26 }
```

Figure 7.12: *Driver Models for Adapter Module*

these container fields accordingly.

Figure 7.12 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 connections:

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

To check the property, use added the following assertion at the end of the `main` method

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

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 unit and previously

written manual stubs. The property did not get violated.

We increased the number of threads in the specification to 3, regenerated the driver, and the property was violated. The counterexample analysis revealed that the property violation occurs 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.

Next, we compared performance of 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 driver was producing a system with a much larger state space than the automated one.

We studied the manual driver: it set one of 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 showed that the manual driver exercised more methods than BEG-generated one. In contrast with the automated driver, 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(name, dbAdapter);
  ...
```

```
  //retrieves
  DbAdapter adapter=(DbAdapter)Naming.lookup(name);
}
```

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 the 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.

Stub Generation

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 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 the original implementation of `Hashtable` and `Vector` without much overhead and for this case study we 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.

Feeding the `DbAdapterImpl` class to BEG, we generated empty stubs at first. For example, a stub for the `currentTimeMillis()` method, used by the `DbAdapter` to figure out connections' age, looks as follows:

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

Treating `DbConnectionImpl` as part of the environment, BEG produces the following code for this class:

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

As described in section 5.5, `TOP_OBJ` fields are used as return values in 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 [48], however, at the time of this experiment, symbolic execution in JPF was available only for small components (e.g., one class)

and we have not applied symbolic execution to our case studies. Without support for abstraction or symbolic execution, `TOP` values are interpreted by the model checker 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. As the next refinement, we added the `DbConnectionImpl` class to the unit, as was suggested by the property, and reran the stub generation preserving side-effects with respect to `DbConnection` objects.

The side-effects analysis in BEG detects allocation sites for unit 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 unit and, for error-free unit, to produce branch coverage similar to the manual environments.

Using `DbAdapterImpl` and `DbConnectionImpl` classes as input, BEG generates 125 classes with 2719 LOC. Stub generation for the adapter unit 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.

Example	Config	States	Trans	Mem	Time	CE	Mds/Ads	Cov_U	Cov_T
$Adapter_e$	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	15,384	32,976	41,344	00:13	516	1.9	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	2,599	4,762	41,088	00:10	99	3.2		
Adapter	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	12,175	26,320	30,272	00:13		2.2	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	2,882	5,903	39,296	00:10		2.3		

Table 7.2: *Verification Results for the Database Adapter Module*

Verification

Table 7.2 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 corrected, *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 unit combined with the Manual $d$rivers and $s$tubs; the $AdMs$ row shows results for Automated $d$rivers combined with the unit and Manual $s$tubs; Ads stands for the use of Automated $d$rivers and $s$tubs; 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, 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. The column CE shows a counterexample length for the faulty unit.

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 unit, 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 unit as the manual drivers, thus, the path coverage for both environments is similar. In addition, we measured *branch coverage*, presented in the last two columns in table 7.2. Column Cov_U shows coverage for branches inside the unit: `DbAdapterImpl` and `DbConnectionImpl` classes; column Cov_T shows total coverage for the entire example. The total coverage shows that the unit closed with the manual environment has 600 branches, the unit closed with automated drivers and manual stubs has 482 branches, and the unit 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 number 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 unit 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 unit suggests that erroneous paths invoke extra code, which is not exercised in the error-free example.

7.3.3 Cache Module

Unit and Property Specification

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 this section, as the entry point into the unit. The class acts as a layer between the main code and the database adapter; it has 1257 LOC and 54 public methods.

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.

Driver Generation

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 described using the following BEG specification:

```
environment{
  driver-assumptions{
    re{
      Main: PDProxy p = makeNew()#
      Writer: edit(); commit(); 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 unit 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 unit

interface. The manual driver could be specified using the following BEG assumptions:

```
environment{
  driver-assumptions{
    re{
      Main:
        PDProxy p1 = makeNew();
        PDProxy p2 = makeNew() #

      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.

Stub Generation

First, using the `PDProxy` class as a unit, we generated empty, stubs. Such stubs were sufficient to find race conditions in the faulty cache module and produce coverage similar to manual drivers 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 unit 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 unit. 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 `UserAgenProxyImpl` class, calculating its effects on the `PDStruct` objects.

BEG detected 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 refined the stub for a database, using an array field to store objects.

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

Verification

Table 7.3 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

Example	Config	States	Trans	Mem	Time	CE	Mds/Ads	Cov_M	Cov_T
$Race_e$	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	636	936	24,640	00:04	317	2.8	28/468	34/538
	MdsP	183	322	37184	00:06	36			
	AdMsP	109	185	27008	00:04	25			
	AdsP	71	114	19456	00:02	24	2.6		
Race	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	1,057	1,690	26,304	00:07		2.2	28/468	36/538
	MdsP	500	948	24,856	00:06				
	AdMsP	287	525	21,944	00:04				
	AdsP	231	420	20,096	00:03		2.2		
$Consist_e$	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	19,617	36,976	50,944	00:18	1197	6.3	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	7,449	13,881	45,184	00:17	441	4.2		
Consist	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	1,503,098	4,108,537	213,888	13:39		16.5	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	407,018	105,4146	182,592	04:25		8.8		

Table 7.3: *Verification Results for the Cache Unit*

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 similar to manual ones. 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.

7.3.4 Discussion

In this section, we present several questions we had before using BEG to verify I-BPM modules.

Is BEG capable of generating environments for I-BPM module? We found BEG specification language capable of specifying the types of drivers that were previously written manually. Also, using BEG stub generation, we generated stubs without 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. We found BEG environments to be more effective than manual ones. 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 aggressively cut parts of the environment that has not influence of the unit.

What is modeled manually and how are these features handled automatically? The following 4 features were modeled specifically during the 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 we checked in this case study.

(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 container for the `UserAgentProxy` class proved to be difficult due to cloning of objects.

(3) Time: The `Date` class was manually stubbed out to always return the same time. BEG generated empty stubs for both the `Date` class and `System.currentTmeMillis()` 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 unit.

What is the impact of POR on performance of environment generation? According to [28], 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 the 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 results 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 section show that BEG environment generation is capable of producing nontrivial reductions on top of POR.

Which BEG methodology is used to generate environments for I-BPM modules? To verify I-BPM modules, we generated concurrent drivers from user specifications, followed by stub generation. First, we generated empty stubs then enhanced them with side-effects to unit data and callbacks. While verification of the two I-BPM modules did not require special handling of J2EE libraries, used to build I-BPM, this case study moved us in the direction of developing a domain-specific methodology for J2EE applications.

7.4 SUN's Pet Store

In this section, we describe our experience with model checking the Pet Store example, which is introduced in section 4.2.1.

7.4.1 Driver Generation: Event-Handling

As described in section 4.2.3, the driver generation produces a driver that simulates user actions on a browser of a web application. There are two ways to approach this task: generate all possible sequences of user actions constrained by a GUI of the web browser or generate drivers according to specifications. In section 7.2, we built a universal driver for GUI applications, reusable across multiple applications. Given the setup code of a GUI application, the universal driver performs all possible sequences of user actions allowed by the GUI.

In web applications, setup is done during the deployment time. A web application usually includes HTML pages that describe a Screen Transition Diagram and there are XML descriptor files that describe events and their corresponding event-handling classes. The setup information can be automatically extracted from HTML and XML artifacts,

```
environment{
  definitions{
  //webActionMap
  createUser = CreateUserHTMLAction
  ...

  //ejbActionMap
  CreateUserEvent = CreateUserEJBAction
  ...
  }
  driver-assumptions{
    re{
      // Set of user scenarios
      Main: createUser;
            createAccount; updateAccount;
           (purchase; remove; (purchase | update))*;
           (purchase; order)*; signOff #
    }
  }
}
```

Figure 7.13: *User Assumptions for Pet Store*

however, we leave this approach to future work, as mentioned in section 8.2. Instead, we specify event-handling mappings and use case scenarios using BEG's specification language.

Figure 7.13 shows a specification for the Pet Store example. The definitions section is used for describing mappings from events to their event-handlers; this information is available in mappings.xml as shown in figure 4.10. There are two levels of event-handling in the Pet Store: through the web and ejb tiers. The assumptions section is written using regular expressions using the event names as atomic actions. Note the use of `Main` as the name of the user thread; the driver is to contain only one main thread.

Figure 7.14 shows excerpts from the driver code generated from described assumptions. The setup section (lines 1-6) instantiates the event handling classes and creates mappings from events to their event-handlers, according to the definitions in the specification. In the assumptions section, each action gets translated into a series of Java statements that create an event (line 9), populate it (lines 11-13) according to analysis described in section 5.4.4, and pass it through the two levels of the event-handling (lines 15-20). The driver generator

```
1   //Setup Section
2   HashMap actionMap = new HashMap();
3   actionMap.put("createUser", CreateUserHTMLAction());
4     ...
5   actionMap.put("CreateUserEvent", new CreateUserEJBAction());
6     ...
7   //Client Events
8   HttpServletRequestImpl createUserEvent =
9                          new HttpServletRequestImpl("createUser");
10  createUserEvent.setParameter("j_username", Abstraction.TOP_STRING);
11  createUserEvent.setParameter("j_password", Abstraction.TOP_STRING);
12  createUserEvent.setParameter("j_password_2", Abstraction.TOP_STRING);
13
14  HTMLAction createUserHTMLHandler = actionMap.get("createUser");
15  Event createUserHTMLResponse = createUserHTMLHandler.perform(createUserEvent);
16
17  String ejbEventName = createUserHTMLResponse.getName();
18  EJBAction createUserEJBHandler = actionMap.get(ejbEventName);
19  createUserEJBHandler.perform(createUserHTMLResponse);
20    ...
```

Figure 7.14: *Driver for Pet Store (excerpts)*

for J2EE applications is customized to generate an event-handling template according to event-handling APIs used in the application.

Appendix A.3.1 shows a sample of the Pet Store specification and automatically generated driver we used in our experiments. By default, BEG generates `TOP` values for all user inputs. We experimented with symbolic execution to calculate sets of interesting concrete values for `TOP` values. Unfortunately, at the time, symbolic execution tools available to us, JPF and Bogor/Kiasan [19], either did not have sufficient support for treatment of `String`s or did not scale beyond one or two symbolic values for one event at a time. We manually refined several `TOP` values to encode nondeterministic choices for the following keys: `j_password`, `j_password_2`, `itemId`, and `itemQuantity`.

7.4.2 Stub Generation: J2EE Components

Using BEG's setting for analysis of J2EE components we generated stubs for all libraries used by the Pet Store example. This step produced 88 packages, 589 classes with 5907 LOC.

```
1  public class SignOnLocalImpl implements SignOnLocal {
2    public static SignOnLocalImpl TOP = new SignOnLocalImpl();
3    SignOnEJB signOnEJB;
4
5    public SignOnLocalImpl(){
6      signOnEJB = new SignOnEJB();
7      try{
8        signOnEJB.ejbCreate();
9      }catch(Exception e){e.printStackTrace();}
10   }
11   public void createUser(java.lang.String param0, java.lang.String param1){
12     signOnEJB.createUser(param0, param1);
13   }
14   public boolean authenticate (String userName, String password){
15     return signOnEJB.authenticate(userName, password);
16   }
17 }
```

Figure 7.15: *SignOn Stub*

In addition to generating library code, there is a need to generate implementations of application-specific classes. A typical J2EE application is a collection of classes, many of them interfaces, which get implemented during deployment time. During stub generation, BEG automatically generates implementations of application-specific interfaces. Figure 7.15 shows a stub for implementation of the local interface of the `SignOnEJB`, used to process the `createUser` event. This step produced 22 packages, 39 classes with 600 LOC.

Some manual refinement was necessary, e.g., implementation of `java.sql.ResultSetImpl` was refined to include a two-dimensional array, populated with several Pet Store items.

7.4.3 Verification Results

We applied JPF to the Pet Store model to validate several types of requirements. In this section we report on checking the following properties:

- *CartEmpty*: A shopping cart becomes empty after placing an order
- *PasswdMatch*: A user should supply matching passwords to create an account
- *Quantity*: Quantity of each item in a cart should be greater than 0

Property	States	Trans	Heap	Mem	Time
CartEmpty	113	314	22642	76	03
PasswdMatch	57	157	11479	75	03
Quantity	79	136	15311	75	03

Table 7.4: *Verification Results for the Pet Store Model*

The above properties were encoded in LTL and checked while running JPF. Table 7.4 shows data for this experiment. JPF reported a violation of the *PasswdMatch* requirement: the user could supply 2 different passwords and still allowed to create an account.

The whole experiment took one month by one person. The bulk of this period was taken by studying the example, its documentation, studying the domain of J2EE applications and tuning BEG to treat J2EE libraries. Code generation by BEG was fast and manual refinements took a couple of days. This experiment convinced researchers at Fujitsu, Japan, to try JPF on their internal web applications. We spent two months setting up environment generation for their internal framework, which uses their own libraries. In the end, we delivered a set of reusable library stubs and driver templates, which could be easily tuned to specific applications within the same framework.

Chapter 8

Conclusion and Future Work

In this Chapter, we conclude by summarizing the contributions of this thesis and describe additional techniques that could enhance BEG as future work.

8.1 Conclusion

In this thesis, we presented environment generation techniques that *enable* model checking of open systems and *reduce* the state space of large closed systems. We presented Bandera Environment Generator, which has automated support for:

- **Interface Discovery:** Given a collection of unit classes, BEG automatically discovers unit and environment interfaces. BEG implements several strategies for discovering the unit interface: for general Java programs, BEG collects all public methods and fields of the unit, for GUI and J2EE applications, BEG collects domain-specific event-handling methods. The unit interface information is used to build universal drivers and check the validity of user assumptions.

  The environment interface, which consists of all external classes, methods, and fields referenced by the unit, is used to define the boundaries of the called environment. This allows BEG to drop many classes from the environment that are not directly referenced by the unit. The behavior of dropped classes is safely approximated by performing static analysis on the actual environment implementation.

- **Specifying Assumptions:** Using method call expressions and assignments as atomic actions, one can describe sequences of actions using regular expressions and LTL. BEG supports specification of three types of data values: concrete, choice, and abstract. In addition, data values can be omitted and BEG can be configured to automatically generate values belonging to one of the categories.

  BEG's specification language was driven by large case studies, including industrial applications. We also found BEG's specification language capable of describing the kinds of drivers that were written previously by hand, as described in section 7.3.

- **Extracting Assumptions:** BEG offers flexibility in how much behavior is to be included in stubs. One can start with empty stubs. If such stubs are not sufficient to find errors, one can turn on side-effects analysis, which can be tuned for different domains to track domain-specific objects. We found that empty stubs enhanced with data effects were capable of uncovering errors. We also found that stub generation could be done without prior knowledge of the applications.

- **Encoding Assumptions into Code:** BEG translates environment assumptions into Java code using special modeling primitives that denote environment nondeterminism. Modeling primitives `randomClass(type)` and `randomReachable(type, ref)` were added to JPF and Bogor to support environment generation.

Measuring Environment Effectiveness: 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. We take a pragmatic approach to measuring quality of environments. We say the environment is effective if it leads to a discovery of an error or, in case of the "verified" result, it shows good branch coverage for a unit under analysis.

Methodology We presented a methodology for environment generation, consisting of: identifying the unit under analysis, writing specifications for drivers, and generating stubs

using BEG's modular side-effects analysis. We studied domains of GUI and J2EE applications and refined our generic methodology to be effective for such domains.

Case Studies We evaluated BEG on a number of case studies, including real industrial case studies from NASA, Fujitsu, and SUN. We found that BEG-generated environments were cost-effective relative to hand-generated environments, as described in sections 7.1 and 7.3.

8.2 Future Work

The environment generation approach presented in this thesis has limitations, which can be addressed in future work:

Automating Unit Identification: In this thesis, the unit is identified manually by a user. In some cases, the domain-specific knowledge is used to identify the unit automatically, e.g., application-specific code for GUI and J2EE applications.

This task can be automated for certain programs using the information produced by static analysis, e.g., one that builds Program Dependence Graph (PDG). Given a property, one can identify the appropriate unit using techniques similar to slicing. Using the PDG, we can set up heuristics to identify classes that influence the property. Since the chains of dependencies may be long, we can set up a bounded search, e.g., include in the unit classes and methods that belong to a bounded-length chain of dependencies for a given property.

Unfortunately, such analyses work best for complete pure Java programs. The case studies we presented in this thesis are open reactive systems, some of them presented as a collection of classes without any dependencies among them, e.g., J2EE applications, which get setup at deployment time.

Specifying Assumptions Extensions: The driver specification language could be extended to handle more Java expressions and statements, including support for property

specification, e.g., assertions. Other, more powerful notations can be used to describe sequences of environment actions, e.g., Context Free Grammars, as described in [42].

Exracting Assumptions Extensions: Several techniques can be used to extract environment assumptions, both statically and dynamically.

- *Static Analysis:* BEG can be extended with other static analysis techniques, e.g., atomicity analysis [29], safe locking analysis [1], domain-partitioning analysis [78].

  BEG can also be combined with slicing, which can be used during stub generation step to increase confidence in results. Once the drivers are generated, off-the-shelf slicers (e.g., the Indus Java slicer[68]) 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 [90]. 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 stub out classes that definitely do not influence the property, then slicing can be applied to the reduced system.

- *Symbolic Execution:* One limitation of the current approach is treatment of `TOP` values, which can be safely interpreted by abstraction engine (e.g., in Bandera [14]) or executed symbolically (e.g., [48]). Unfortunately, at the time of our case studies, symbolic execution in JPF worked only for small units, and due to scalability issues, we did not use automated support to treat the `TOP` values according to their semantics. This limitation can be addressed by experimenting with the latest symbolic execution in JPF or using test case generation to refine `TOP` values.

- *Run-time Analysis:* Run-time analysis can be used to extract environment assumptions after observing many runs of the program under analysis. There are many run-time analysis tools for Java, e.g., JPax [39], Sofya [49]. A typical run-time analysis

methodology includes instrumenting the code, e.g., marking actions that environment may perform on the unit; running the program multiple times, recoding the traces and then processing them to learn usage patterns. One way to process the traces is use off-the-shelf PFSA learners [67] that generalize from a set of traces and discover likely automata-based properties [3]. This approach can be used to learn patterns of environment actions, however, this technique assumes that the unit-environment system is set up and running.

- *Learning Assumptions:* Learning algorithms [2, 12] can also be used to learn environment assumptions.

Extending Support for the J2EE Framework: As mentioned in sections 4.2.3 and 7.4, information about screen transitions for J2EE applications is usually encoded using non-Java artifacts, e.g., HTML. We can develop automated support for generating Java code for screen transitions and develop universal drivers, similar to the ones we developed for GUI applications. Then, we can check all use case scenarios enabled by a screen transition of a given J2EE application.

Extending to Other Frameworks: The landscape of frameworks used to develop applications is always changing. In this thesis, we studied Swing/AWT and J2EE frameworks, however, BEG can be extended to handle other frameworks. For example, for web applications, BEG can be extended to handle frameworks used to process user input at the front end of web applications, e.g., Struts [80], and frameworks used to interface with a database, e.g., Hibernate [40].

Automating Environment Refinement: In this work, we manually went through several refinements of the unit-environment system. For example, as described in section 7.3, based on model checking results, we adjusted modules to include classes that were tightly coupled with the unit; 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 units, drivers, and stubs based on results of model checking and coverage information.

Bibliography

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[2] 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.

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[4] 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.

[5] 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.

[6] E. Barlas and T. Bultan. Netstub: a framework for verification of distributed java applications. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 24–33, New York, NY, USA, 2007. ACM.

[7] 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.

[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates, 2002.

[9] 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.

[10] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, 1993.

[11] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[12] 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.

[13] 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.

[14] 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.

[15] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera specification language. *International Journal on Software Tools for Technology Transfer*, 4(1):34–56, 2002.

[16] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference*

Record of the Fourth Annual ACM Symposium on Principles of Programming Languages, pages 238–252, 1977.

[17] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software – Practice and Experience*, 34(11):1025–1050, Sept. 2004.

[18] A. Degani and M. Heymann. Formal verification of human-automation interaction. *Human Factors*, 2002.

[19] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE*, pages 157–166, 2006.

[20] X. Deng, Robby, and J. Hatcliff. Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems, 2007.

[21] L. Dillon, G. Kutty, L. Moser, P. Melliar-Smith, and Y. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[22] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. P. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *TACAS*, pages 73–89, 2006.

[23] 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.

[24] M. B. Dwyer and C. S. Păsăreanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 1998.

[25] M. B. Dwyer and C. S. Păsăreanu. Model checking generic container implementations. In *Proceedings of the 1st Symposium on Generic Programming*, May 1998.

[26] 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.

[27] FindBugs. Website. http://findbugs.sourceforge.net.

[28] C. Flanagan and S. Freund. Atomizer: A dynamic atomocity checker for multithreaded programs. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, 2004.

[29] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM Press.

[30] C. Flanagan and S. Qadeer. Thread modular model checking. In *Model Checking Software (LNCS 2648)*, May 2003.

[31] S. Framework. Website. http://www.springframework.org/.

[32] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. Mock roles, objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246, New York, NY, USA, 2004. ACM.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[34] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[35] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Jan. 1997.

[36] 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.

[37] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[38] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 13(4), 2000.

[39] K. Havelund and G. Rosu. Monitoring java programs with java pathexplorer. In *In Proceedings of Runtime Verification (RV01*, pages 97–114. Elsevier, 2001.

[40] Hibernate. Website. http://www.hibernate.org/.

[41] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[42] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 39–49, New York, NY, USA, 2007. ACM.

[43] G. Hughes, S. P. Rajan, T. Sidle, and K. Swenson. Error detection in concurrent java programs. In *Workshop on Software Model Checking*, 2005.

[44] JLint. Website. http://artho.com/jlint.

[45] JPF. Website. http://javapathfinder.sourceforge.net.

[46] JTest. Website. http://www.parasoft.com.

[47] JUnit. Website. http://www.junit.org.

[48] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[49] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting rapid development of dynamic program analyses for java. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 51–52, Washington, DC, USA, 2007. IEEE Computer Society.

[50] O. Kupferman and M. Y. Vardi. Modular model checking. In *COMPOS (LNCS 1536)*, 1998.

[51] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. *ACM SIGPLAN Notices*, 28(6):56–67, 1993.

[52] G. T. Leavens, A. L. Baker, and C. Ruby. Jml: a java modeling language. In *In Formal Underpinnings of Java Workshop (at OOPSLA'98*, 1998.

[53] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[54] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. *Lecture Notes in Computer Science*, 2126:279, 2001.

[55] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for java. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 73–79, 2001.

[56] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.

[57] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992.

[58] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[59] M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *The First Symposium on Networked Systems Design and Implementation*, 2004.

[60] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[61] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 815–816, New York, NY, USA, 2007. ACM.

[62] E. Palmer. Oops, it didn't arm: a case study of two automation surprises. In *Proceedings of the Eighth International Symposium on Aviation Psychology*, 1995.

[63] P. Parizek and F. Plasil. Specification and generation of environment for model checking of software components. *Electron. Notes Theor. Comput. Sci.*, 176(2):143–154, 2007.

[64] 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.

[65] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1985.

[66] C. S. Păsăreanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software : A comparative case study. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 16 80)*, Sept. 1999.

[67] A. Raman and J. Patrick. The sk-strings method for inferring pfsa. In *Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on machine learning (ICML97)*, 1997.

[68] V. Ranganath. Indus Website. http://indus.projects.cis.ksu.edu.

[69] C. Razafimahefa. A study of side-effect analyses for java. Master's thesis, McGill University, Dec. 1999.

[70] Robby. Bogor Website. http://bogor.projects.cis.ksu.edu, 2003.

[71] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

[72] E. Rodrguez, M. B. Dwyer, and J. Hatcliff. flexible framework for the estimation of coverage metrics in explicit state software model checking. In *in Proceedings of the 2004 International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, 2004.

[73] 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.

[74] L. Sherry, M. Feary, P. Polson, and E. Palmer. Autopilot tutor: Building and maintaining autopilot skills.

[75] Soot. Tutorial. http://www.sable.mcgill.ca/soot/tutorial/index.html.

[76] Soot. Website. http://www.sable.mcgill.ca/soot/.

[77] 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.

[78] 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.

[79] 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.

[80] Struts. Website. http://struts.apache.org/.

[81] SUN. J2EE 1.4 Tutorial. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.

[82] SUN. Java Pet Store. http://java.sun.com/j2ee/1.4/download.htmlsamples.

[83] SUN. Swing Tutorial. http://java.sun.com/docs/books/tutorial/uiswing/index.html.

[84] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[85] O. Tkachuk. Adapting side effects analysis for modular program model checking. Master's thesis, Kansas State University, 2003.

[86] O. Tkachuk, G. Brat, and W. Visser. Using code level model checking to discover automation surprises. In *Proceedings of the 2002 Digital Avionics Systems Conference*, 2002.

[87] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of the Fourth joint meeting of the European Software*

Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Sept. 2003.

[88] 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.

[89] O. Tkachuk and S. P. Rajan. Application of automated environment generation to commercial software. In *ISSTA'06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 203–214, New York, NY, USA, 2006. ACM Press.

[90] O. Tkachuk and S. P. Rajan. Combining environment generation and slicing for modular software model checking. In *ASE*, pages 401–404, 2007.

[91] R. Valle-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON'99*, Nov. 1999.

[92] 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.

[93] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms For The Construction And Analysis Of Systems (TACAS '05)*, April 2005.

[94] 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.

[95] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and*

Applications, volume 34(10) of *ACM SIGPLAN Notices*, pages 187–206, Denver, CO, Oct. 1999. ACM Press.

[96] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, April 2005.

[97] 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.

[98] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Appendix A

BEG Configurations and Generated Code

A.1 Observer-Observable

A.1.1 Universal Driver

BEG configuration to generate a universal driver:

```
ApplInfo = applinfo.domain.DeafultDriverInfo
InterfaceFinder = applinfo.UnitInterfaceFinder
CodeGenerator = codegen.UniversalDriverGenerator
numThreads = 2
numInstances = 2
unit = Subject Watcher
```

The universal driver generated automatically by BEG:

```
1  public class EnvDriver {
2    public static void main(String[] param1){
3      Subject s = new Subject();
4      Subject s2 = new Subject();
5      Watcher w = new Watcher();
6      Watcher w2 = new Watcher();
7      new EnvDriverThread(s, s2, w, w2).start();
8      new EnvDriverThread(s, s2, w, w2).start();
9    }
10 }
```

```
1  public class EnvDriverThread extends java.lang.Thread {
2    public Subject s, s2;
3    public Watcher w, w2;
4    public EnvDriverThread(Subject param1, Subject param2,
5                           Watcher param3, Watcher param4){
6      s = param1;
7      s2 = param2;
```

```
8       w = param3;
9       w2 = param4;
10    }
11    public void run(){
12      while(true){
13        int choice = Verify.random(8);
14        switch(choice){
15            case 0:
16                ((Watcher)Verify.randomObject("Watcher")).registered =
17                  Abstraction.TOP_BOOL;
18                  break;
19            case 1:
20                  ((Watcher)Verify.randomObject("Watcher")).
21                    update(Verify.randomObject("Observable"),
22                    Verify.randomObject("Object"));
23                  break;
24            case 2:
25                  ((Subject)Verify.randomObject(Subject)).
26                  delete((Watcher)Verify.randomObject("Watcher")));
27                  break;
28            ...
29            case 8:
30                  ((Subject)Verify.randomObject(Subject)).
31                  add(((Watcher)Verify.randomObject("Watcher")));
32                  break;
33            }
34      }
35  }
```

The universal driver corresponds to the following BEG specification:

```
environment {
  setup { 2 Subject; 2 Watcher }
  driver-assumptions {
      re{
        2 EnvDriverThread:
          (changeState() | add() | delete() | hasChanged() |
          setChanged() | notifyObservers() | update() |
          registered = TOP_BOOL )*
      }
  }
}
```

A.1.2 User Specified Stubs

BEG configuration to generate stubs from user specifications:

```
ApplInfo = applinfo.domain.DefaultStubInfo
InterfaceFinder = applinfo.EnvInterfaceFinder
AssumptionsAcquirer = spec.SpecReader
```

```
CodeGenerator = codegen.SpecStubGenerator
specFileName = specs/observer-stubs.spec
unit = Subject Watcher
```

Stub specification sample:

```
environment {
  stubs-assumptions {
    Buffer{
      re removeFirst(){
        return (Watcher)Verify.randomReachable("Watcher", this);
      }
      re register(Watcher p){
        p.registered = true;
      }
      //the rest of the methods are identified and stubbed out by BEG
    }
  }
}
```

A.1.3 Empty Stubs

BEG configuration to generate empty stubs:

```
ApplInfo = applinfo.domain.DefaultStubInfo
InterfaceFinder = applinfo.EnvInterfaceFinder
CodeGenerator = codegen.EmptyStubGenerator
unit = Subject Watcher
```

```
1  import gov.nasa.jpf.jvm.Verify;
2  import edu.ksu.cis.bandera.Abstraction;
3  public class Buffer{
4    public static Buffer TOP_OBJ = new Buffer();
5
6    public Buffer(){
7    }
8    public void register(Watcher param0){
9    }
10   public void unregister(Watcher param0){
11   }
12   public void copy(Buffer param0){
13   }
14   public Watcher removeFirst(){
15     return ((Watcher)Verify.randomObject("Watcher"));
16   }
17   public boolean isEmpty(){
18     return Abstraction.TOP_BOOL;
19   }
20 }
```

```
1  package java.util;
2  import gov.nasa.jpf.jvm.Verify;
3  import edu.ksu.cis.bandera.Abstraction;
4
5  public interface Observer {
6
7  }
```

```
1  package java.util;
2  import gov.nasa.jpf.jvm.Verify;
3  import edu.ksu.cis.bandera.Abstraction;
4
5  public class Observable {
6    public static java.util.Observable TOP_OBJ =
7           new java.util.Observable();
8
9    public Observable(){
10   }
11 }
```

A.2 GUI Examples

A.2.1 Universal Driver

Reusable universal driver:

```
1  package env;
2
3  import env.java.awt.*;
4  import env.java.awt.event.*;
5  import env.javax.swing.*;
6  import env.javax.swing.event.*;
7  import env.java.beans.*;
8  import env.java.util.EventListener;
9  import java.util.Vector;
10 import gov.nasa.jpf.jvm.Verify;
11
12 public class UserModel {
13   public static Window chooseTopWindow() {
14     Window window = null;
15     System.out.println("Fetching␣a␣top␣level␣window");
16
17     Vector modalDialogs = SwingUtilities.getModalDialogs();
18     Dialog modal;
19
20     if (!modalDialogs.isEmpty()) {
21       modal = (Dialog) modalDialogs.lastElement();
22           if (modal != null && modal.isVisible()) {
23             window = modal;
```

```
24            }
25      }
26      System.out.println("Fetching␣a␣frame␣␣or␣non-modal␣dialog");
27      if (window == null) {
28            Vector topWindows = SwingUtilities.getTopWindows();
29            window = (Window) Verify.randomReachable("env.java.awt.Window",
30                                    topWindows);
31
32          }
33      return window;
34    }
35    public static void notifyListeners(JComponent container) {
36
37      EventListener[] list = container.getListeners();
38
39      EventListener listener;
40      for (int i = 0; i < list.length; i++) {
41        listener = list[i];
42            if (listener == null)
43              continue;
44            if (listener instanceof ChangeListener) {
45              ((ChangeListener) listener).stateChanged(new ChangeEvent(
46                                                    container));
47            }
48            if (listener instanceof ItemListener) {
49              ((ItemListener) listener).itemStateChanged(new ItemEvent(
50                                                    container));
51            }
52            if (listener instanceof ActionListener) {
53             ((ActionListener) listener).actionPerformed(new ActionEvent(
54                                                    container));
55            }
56            if (listener instanceof PropertyChangeListener) {
57             ((PropertyChangeListener) listener)
58                          .propertyChange(new PropertyChangeEvent(container));
59
60            }
61        ...
62      }
63    }
64  }
```

Reusable model of the class that keeps track of modal and non-modal dialogs:

```
1  package env.javax.swing;
2
3  import gov.nasa.jpf.jvm.Verify;
4  import edu.ksu.cis.bandera.Abstraction;
5  import java.util.Vector;
6  import env.java.awt.*;
7
8  public class SwingUtilities {
```

```
9    public static Vector topWindows = new Vector();
10
11   private static int maxNumWindows = 10;
12
13   public static Vector modalDialogs = new Vector();
14
15   public static void invokeLater(java.lang.Runnable param0) {
16   }
17   public static void updateComponentTreeUI(env.java.awt.Component param0) {
18   }
19
20   //auxiliary methods to keep track of top windows
21
22   public static void addTopWindow(Window window) {
23     if (topWindows.size() < maxNumWindows) {
24           topWindows.add(window);
25         }
26   }
27   public static void removeTopWindow(Window window) {
28         topWindows.remove(window);
29   }
30   public static Vector getTopWindows() {
31     return topWindows;
32   }
33   public static void addModalDialog(Dialog dialog) {
34     modalDialogs.add(dialog);
35   }
36   public static void removeModalDialog(Dialog dialog) {
37         modalDialogs.remove(dialog);
38   }
39   public static Vector getModalDialogs() {
40         return modalDialogs;
41   }
42 }
```

Application-specific driver built on top of the reusable universal driver:

```
1  import java.util.Vector.*;
2
3  import env.java.UserModel;
4  import env.java.util.EventListener;
5  import env.javax.swing.*;
6  import env.java.awt.*;
7  import env.javax.swing.event.*;
8  import env.java.awt.event.*;
9  import env.java.beans.*;
10
11 import gov.nasa.jpf.jvm.Verify;
12
13 public class EnvDriverDialogDemo{
14
15   public static void main(String[] args) {
```

```
16      // Create and set up the window
17      JFrame.setDefaultLookAndFeelDecorated(true);
18      JFrame frame = new JFrame("DialogDemo");
19      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20
21      // Create and set up the content pane.
22      DialogDemo newContentPane = new DialogDemo(frame);
23      // content panes must be opaque
24      newContentPane.setOpaque(true);
25      frame.setContentPane(newContentPane);
26
27      // Display the window
28      frame.pack();
29      frame.setVisible(true);
30      SwingUtilities.addTopWindow(frame);
31
32      // event-handling loop
33      Window window;
34      JComponent container;
35
36      while (true) {
37        window = UserModel.chooseTopWindow();
38
39        container = (JComponent) Verify.randomReachable(
40                    "env.javax.swing.JComponent", window);
41
42        UserModel.notifyListeners(container);
43      }
44    }
45  }
```

A.3 SUN's Pet Store

A.3.1 User Specified Driver

BEG configuration file for driver generation:

```
ApplInfo = applinfo.domain.J2EEDriverInfo
InterfaceFinder = applinfo.UnitInterfaceFinder
AssumptionsAcquirer = spec.SpecReader
CodeGenerator = codegen.J2EESpecDriverGenerator
specFileName = specs/petstore.spec
printActions = true
unit = com.sun.j2ee.blueprints.petstore.controller.web.actions.*
com.sun.j2ee.blueprints.petstore.controller.ejb.actions.*
com.sun.j2ee.blueprints.petstore.controller.events.*
```

Specification files for driver generation:

```
environment{
  definitions{
    //webActionMap
    createUser = CreateUserHTMLAction
    createAccount =  CustomerHTMLAction
    updateAccount =  CustomerHTMLAction
    purchase =  CartHTMLAction
    remove =  CartHTMLAction
    update = CartHTMLAction
    order = OrderHTMLAction
    signOff =  SignOffHTMLAction

    //ejbActionMap
    CartEvent = CartEJBAction
    CreateUserEvent = CreateUserEJBAction
    CustomerEvent = CustomerEJBAction
    OrderEvent =  OrderEJBAction
    SignOnEvent = SignOnEJBAction
  }
  driver-assumptions{
    re{
      // Set of user scenarios
      Main: createUser;
                createAccount; (createAccount | updateAccount);
                (purchase; remove; (purchase | update)) ^ {0,2};
                (purchase; order)^{0,3};
                 signOff #
    }
  }
}
```

Driver automatically generated from the above specification (with `Abs` used for `Abstraction` and some imports factored out for reduction of the code):

```
1  import gov.nasa.jpf.jvm.Verify;
2  import edu.ksu.cis.bandera.Abs;
3  import com.sun.j2ee.blueprints.petstore.controller.web.actions.*;
4  import com.sun.j2ee.blueprints.petstore.controller.ejb.actions.*;
5  import com.sun.j2ee.blueprints.petstore.controller.events.*;
6  import com.sun.j2ee.blueprints.waf.controller.web.action.*;
7  import com.sun.j2ee.blueprints.waf.controller.ejb.action.*;
8  import com.sun.j2ee.blueprints.waf.event.*;
9  import env.javax.servlet.http.*;
10
11 public class EnvDriver {
12   public static void main(java.lang.String[] param0){
13     java.util.HashMap actionMap = new java.util.HashMap();
14     actionMap.put("purchase", new CartHTMLAction());
15     actionMap.put("update", new CartHTMLAction());
16     actionMap.put("createAccount", new CustomerHTMLAction());
17     actionMap.put("order", new OrderHTMLAction());
```

```
18      actionMap.put("signOff", new SignOffHTMLAction());
19      actionMap.put("updateAccount", new CustomerHTMLAction());
20      actionMap.put("createUser", new CreateUserHTMLAction());
21      actionMap.put("remove", new CartHTMLAction());
22
23      actionMap.put("SignOnEvent", new SignOnEJBAction());
24      actionMap.put("OrderEvent", new OrderEJBAction());
25      actionMap.put("CartEvent", new CartEJBAction());
26      actionMap.put("CustomerEvent", new CustomerEJBAction());
27      actionMap.put("CreateUserEvent", new CreateUserEJBAction());
28      EnvUserDemo EnvUser0 = new EnvUserDemo(actionMap, actionMap);
29
30      try{
31        HttpServletRequestImpl createUserEvent =
32          new HttpServletRequestImpl("createUser");
33        createUserEvent.setParameter("j_password", Abs.TOP_STRING);
34        createUserEvent.setParameter("j_password_2", Abs.TOP_STRING);
35        createUserEvent.setParameter("j_username", Abs.TOP_STRING);
36        HTMLAction createUserHTMLHandler =
37          (HTMLAction)actionMap.get("createUser");
38        if(createUserHTMLHandler!=null){
39          System.out.println("@EnvDriverThread:␣createUser");
40          Event createUserHTMLResponse =
41            createUserHTMLHandler.perform(createUserEvent);
42          if(createUserHTMLResponse!=null){
43            EJBAction createUserEJBHandler =
44            (EJBAction)actionMap.get(createUserHTMLResponse.getClass().getName());
45            if(createUserEJBHandler!=null){
46              EventResponse createUserEJBResponse =
47                createUserEJBHandler.perform(createUserHTMLResponse);
48            }
49          }
50        }
51      }
52      catch(Exception e ){
53        e.printStackTrace();
54      }
55      try{
56        HttpServletRequestImpl createAccountEvent =
57          new HttpServletRequestImpl("createAccount");
58        createAccountEvent.setParameter("email_a", Abs.TOP_STRING);
59        createAccountEvent.setParameter("mylist_on", Abs.TOP_STRING);
60        createAccountEvent.setParameter("banners_on", Abs.TOP_STRING);
61        createAccountEvent.setParameter("credit_card_expiry_year", Abs.TOP_STRING);
62        createAccountEvent.setParameter("telephone_number_a", Abs.TOP_STRING);
63        createAccountEvent.setParameter("country_a", Abs.TOP_STRING);
64        createAccountEvent.setParameter("given_name_a", Abs.TOP_STRING);
65        createAccountEvent.setParameter("city_a", Abs.TOP_STRING);
66        createAccountEvent.setParameter("family_name_a", Abs.TOP_STRING);
67        createAccountEvent.setParameter("favorite_category", Abs.TOP_STRING);
68        createAccountEvent.setParameter("address_1_a", Abs.TOP_STRING);
```

```
69        createAccountEvent.setParameter("language", Abs.TOP_STRING);
70        createAccountEvent.setParameter("state_or_province_a", Abs.TOP_STRING);
71        createAccountEvent.setParameter("credit_card_number", Abs.TOP_STRING);
72        createAccountEvent.setParameter("address_2_a", Abs.TOP_STRING);
73        createAccountEvent.setParameter("postal_code_a", Abs.TOP_STRING);
74        createAccountEvent.setParameter("credit_card_expiry_month", Abs.TOP_STRING);
75        createAccountEvent.setParameter("credit_card_type", Abs.TOP_STRING);
76        HTMLAction createAccountHTMLHandler =
77          (HTMLAction)actionMap.get("createAccount");
78        if(createAccountHTMLHandler!=null){
79          System.out.println("@EnvDriverThread:␣createAccount");
80          Event createAccountHTMLResponse =
81            createAccountHTMLHandler.perform(createAccountEvent);
82          if(createAccountHTMLResponse!=null){
83            EJBAction createAccountEJBHandler =
84            (EJBAction)actionMap.get(createAccountHTMLResponse.getClass().getName());
85            if(createAccountEJBHandler!=null){
86              EventResponse createAccountEJBResponse =
87                createAccountEJBHandler.perform(createAccountHTMLResponse);
88            }
89          }
90        }
91      }
92      catch(Exception e ){
93        e.printStackTrace();
94      }
95      int choice4=Verify.random(1);
96      switch(choice4){
97      case 0:
98        try{
99          HttpServletRequestImpl updateAccountEvent =
100           new HttpServletRequestImpl("updateAccount");
101       updateAccountEvent.setParameter("email_a", Abs.TOP_STRING);
102       updateAccountEvent.setParameter("mylist_on", Abs.TOP_STRING);
103       updateAccountEvent.setParameter("banners_on", Abs.TOP_STRING);
104       updateAccountEvent.setParameter("credit_card_expiry_year", Abs.TOP_STRING);
105       updateAccountEvent.setParameter("telephone_number_a", Abs.TOP_STRING);
106       updateAccountEvent.setParameter("country_a", Abs.TOP_STRING);
107       updateAccountEvent.setParameter("given_name_a", Abs.TOP_STRING);
108       updateAccountEvent.setParameter("city_a", Abs.TOP_STRING);
109       updateAccountEvent.setParameter("family_name_a", Abs.TOP_STRING);
110       updateAccountEvent.setParameter("favorite_category", Abs.TOP_STRING);
111       updateAccountEvent.setParameter("address_1_a", Abs.TOP_STRING);
112       updateAccountEvent.setParameter("language", Abs.TOP_STRING);
113       updateAccountEvent.setParameter("state_or_province_a", Abs.TOP_STRING);
114       updateAccountEvent.setParameter("credit_card_number", Abs.TOP_STRING);
115       updateAccountEvent.setParameter("address_2_a", Abs.TOP_STRING);
116       updateAccountEvent.setParameter("postal_code_a", Abs.TOP_STRING);
117       updateAccountEvent.setParameter("credit_card_expiry_month", Abs.TOP_STRING);
118       updateAccountEvent.setParameter("credit_card_type", Abs.TOP_STRING);
119       HTMLAction updateAccountHTMLHandler =
```

```
120           (HTMLAction)actionMap.get("updateAccount");
121         if(updateAccountHTMLHandler!=null){
122         System.out.println("@EnvDriverThread:␣updateAccount");
123         Event updateAccountHTMLResponse =
124           updateAccountHTMLHandler.perform(updateAccountEvent);
125           if(updateAccountHTMLResponse!=null){
126             EJBAction updateAccountEJBHandler =
127             (EJBAction)actionMap.get(updateAccountHTMLResponse.getClass().getName());
128             if(updateAccountEJBHandler!=null){
129               EventResponse updateAccountEJBResponse =
130               updateAccountEJBHandler.perform(updateAccountHTMLResponse);
131             }
132           }
133         }
134       }
135       catch(Exception e ){
136       e.printStackTrace();
137       }
138       break;
139     case 1:
140       try{
141         HttpServletRequestImpl createAccountEvent =
142           new HttpServletRequestImpl("createAccount");
143         createAccountEvent.setParameter("email_a", Abs.TOP_STRING);
144         createAccountEvent.setParameter("mylist_on", Abs.TOP_STRING);
145         createAccountEvent.setParameter("banners_on", Abs.TOP_STRING);
146         createAccountEvent.setParameter("credit_card_expiry_year", Abs.TOP_STRING);
147         createAccountEvent.setParameter("telephone_number_a", Abs.TOP_STRING);
148         createAccountEvent.setParameter("country_a", Abs.TOP_STRING);
149         createAccountEvent.setParameter("given_name_a", Abs.TOP_STRING);
150         createAccountEvent.setParameter("city_a", Abs.TOP_STRING);
151         createAccountEvent.setParameter("family_name_a", Abs.TOP_STRING);
152         createAccountEvent.setParameter("favorite_category", Abs.TOP_STRING);
153         createAccountEvent.setParameter("address_1_a", Abs.TOP_STRING);
154         createAccountEvent.setParameter("language", Abs.TOP_STRING);
155         createAccountEvent.setParameter("state_or_province_a", Abs.TOP_STRING);
156         createAccountEvent.setParameter("credit_card_number", Abs.TOP_STRING);
157         createAccountEvent.setParameter("address_2_a", Abs.TOP_STRING);
158         createAccountEvent.setParameter("postal_code_a", Abs.TOP_STRING);
159         createAccountEvent.setParameter("credit_card_expiry_month", Abs.TOP_STRING);
160         createAccountEvent.setParameter("credit_card_type", Abs.TOP_STRING);
161         HTMLAction createAccountHTMLHandler =
162           (HTMLAction)actionMap.get("createAccount");
163         if(createAccountHTMLHandler!=null){
164           System.out.println("@EnvDriverThread:␣createAccount");
165           Event createAccountHTMLResponse =
166             createAccountHTMLHandler.perform(createAccountEvent);
167           if(createAccountHTMLResponse!=null){
168             EJBAction createAccountEJBHandler =
169             (EJBAction)actionMap.get(createAccountHTMLResponse.getClass().getName());
170             if(createAccountEJBHandler!=null){
```

```
171               EventResponse createAccountEJBResponse =
172               createAccountEJBHandler.perform(createAccountHTMLResponse);
173             }
174           }
175         }
176       }
177       catch(Exception e ){
178       e.printStackTrace();
179       }
180       break;
181     }
182     for(int i=0;i<0+Verify.random(3);++i){
183       try{
184         HttpServletRequestImpl purchaseEvent =
185           new HttpServletRequestImpl("purchase");
186         purchaseEvent.setParameter("itemId",  Abs.TOP_STRING);
187         HTMLAction purchaseHTMLHandler = (HTMLAction)actionMap.get("purchase");
188         if(purchaseHTMLHandler!=null){
189           System.out.println("@EnvDriverThread:␣purchase");
190           Event purchaseHTMLResponse =
191             purchaseHTMLHandler.perform(purchaseEvent);
192           if(purchaseHTMLResponse!=null){
193             EJBAction purchaseEJBHandler =
194             (EJBAction)actionMap.get(purchaseHTMLResponse.getClass().getName());
195             if(purchaseEJBHandler!=null){
196               EventResponse purchaseEJBResponse =
197                 purchaseEJBHandler.perform(purchaseHTMLResponse);
198             }
199           }
200         }
201       }
202       catch(Exception e ){
203         e.printStackTrace();
204       }
205       try{
206         HttpServletRequestImpl removeEvent = new
207           HttpServletRequestImpl("remove");
208         removeEvent.setParameter("itemId",  Abs.TOP_STRING );
209         HTMLAction removeHTMLHandler = (HTMLAction)actionMap.get("remove");
210         if(removeHTMLHandler!=null){
211           System.out.println("@EnvDriverThread:␣remove");
212           Event removeHTMLResponse = removeHTMLHandler.perform(removeEvent);
213           if(removeHTMLResponse!=null){
214             EJBAction removeEJBHandler =
215             (EJBAction)actionMap.get(removeHTMLResponse.getClass().getName());
216             if(removeEJBHandler!=null){
217               EventResponse removeEJBResponse =
218                 removeEJBHandler.perform(removeHTMLResponse);
219             }
220           }
221         }
```

```
222     }
223     catch(Exception e ){
224     e.printStackTrace();
225     }
226     int choice7=Verify.random(1);
227     switch(choice7){
228     case 0:
229       try{
230         HttpServletRequestImpl updateEvent = new
231           HttpServletRequestImpl("update");
232         updateEvent.setParameter("itemId",  Abs.TOP_STRING );
233         updateEvent.setParameter("itemQuantity", Abs.TOP_STRING);
234         HTMLAction updateHTMLHandler = (HTMLAction)actionMap.get("update");
235         if(updateHTMLHandler!=null){
236           System.out.println("@EnvDriverThread:␣update");
237           Event updateHTMLResponse = updateHTMLHandler.perform(updateEvent);
238           if(updateHTMLResponse!=null){
239             EJBAction updateEJBHandler =
240             (EJBAction)actionMap.get(updateHTMLResponse.getClass().getName());
241             if(updateEJBHandler!=null){
242               EventResponse updateEJBResponse =
243                 updateEJBHandler.perform(updateHTMLResponse);
244             }
245           }
246         }
247       }
248       catch(Exception e ){
249       e.printStackTrace();
250       }
251       break;
252     case 1:
253       try{
254         HttpServletRequestImpl purchaseEvent = new
255         HttpServletRequestImpl("purchase");
256         purchaseEvent.setParameter("itemId",  Abs.TOP_STRING);
257         HTMLAction purchaseHTMLHandler = (HTMLAction)actionMap.get("purchase");
258         if(purchaseHTMLHandler!=null){
259           System.out.println("@EnvDriverThread:␣purchase");
260           Event purchaseHTMLResponse =
261           purchaseHTMLHandler.perform(purchaseEvent);
262           if(purchaseHTMLResponse!=null){
263             EJBAction purchaseEJBHandler =
264             (EJBAction)actionMap.get(purchaseHTMLResponse.getClass().getName());
265             if(purchaseEJBHandler!=null){
266               EventResponse purchaseEJBResponse =
267               purchaseEJBHandler.perform(purchaseHTMLResponse);
268             }
269           }
270         }
271       }
272       catch(Exception e ){
```

```
273         e.printStackTrace();
274         }
275         break;
276         }
277       }
278
279       for(int i=0;i<0+Verify.random(3);++i){
280         try{
281           HttpServletRequestImpl purchaseEvent = new
282             HttpServletRequestImpl("purchase");
283           purchaseEvent.setParameter("itemId", Abs.TOP_STRING);
284           HTMLAction purchaseHTMLHandler = (HTMLAction)actionMap.get("purchase");
285           if(purchaseHTMLHandler!=null){
286             System.out.println("@EnvDriverThread: purchase");
287             Event purchaseHTMLResponse =
288               purchaseHTMLHandler.perform(purchaseEvent);
289             if(purchaseHTMLResponse!=null){
290               EJBAction purchaseEJBHandler =
291              (EJBAction)actionMap.get(purchaseHTMLResponse.getClass().getName());
292               if(purchaseEJBHandler!=null){
293                 EventResponse purchaseEJBResponse =
294                   purchaseEJBHandler.perform(purchaseHTMLResponse);
295               }
296             }
297           }
298         }
299         catch(Exception e ){
300           e.printStackTrace();
301         }
302         try{
303           HttpServletRequestImpl orderEvent = new
304             HttpServletRequestImpl("order");
305           orderEvent.setParameter("address_2_b", Abs.TOP_STRING);
306           orderEvent.setParameter("email_a", Abs.TOP_STRING);
307           orderEvent.setParameter("city_b", Abs.TOP_STRING);
308           orderEvent.setParameter("telephone_number_a", Abs.TOP_STRING);
309           orderEvent.setParameter("country_a", Abs.TOP_STRING);
310           orderEvent.setParameter("family_name_b", Abs.TOP_STRING);
311           orderEvent.setParameter("postal_code_b", Abs.TOP_STRING);
312           orderEvent.setParameter("given_name_a", Abs.TOP_STRING);
313           orderEvent.setParameter("address_1_b", Abs.TOP_STRING);
314           orderEvent.setParameter("city_a", Abs.TOP_STRING);
315           orderEvent.setParameter("family_name_a", Abs.TOP_STRING);
316           orderEvent.setParameter("given_name_b", Abs.TOP_STRING);
317           orderEvent.setParameter("country_b", Abs.TOP_STRING);
318           orderEvent.setParameter("address_1_a", Abs.TOP_STRING);
319           orderEvent.setParameter("state_or_province_a", Abs.TOP_STRING);
320           orderEvent.setParameter("address_2_a", Abs.TOP_STRING);
321           orderEvent.setParameter("postal_code_a", Abs.TOP_STRING);
322           orderEvent.setParameter("state_or_province_b", Abs.TOP_STRING);
323           orderEvent.setParameter("telephone_number_b", Abs.TOP_STRING);
```

```
324           orderEvent.setParameter("email_b", Abs.TOP_STRING);
325           HTMLAction orderHTMLHandler = (HTMLAction)actionMap.get("order");
326           if(orderHTMLHandler!=null){
327             System.out.println("@EnvDriverThread:␣order");
328             Event orderHTMLResponse =
329               orderHTMLHandler.perform(orderEvent);
330             if(orderHTMLResponse!=null){
331               EJBAction orderEJBHandler =
332               (EJBAction)actionMap.get(orderHTMLResponse.getClass().getName());
333               if(orderEJBHandler!=null){
334                 EventResponse orderEJBResponse =
335                   orderEJBHandler.perform(orderHTMLResponse);
336               }
337             }
338           }
339         }
340         catch(Exception e ){
341           e.printStackTrace();
342         }
343       }
344       try{
345         HttpServletRequestImpl signOffEvent = new
346           HttpServletRequestImpl("signOff");
347         HTMLAction signOffHTMLHandler = (HTMLAction)actionMap.get("signOff");
348         if(signOffHTMLHandler!=null){
349           System.out.println("@EnvDriverThread:␣signOff");
350           Event signOffHTMLResponse =
351             signOffHTMLHandler.perform(signOffEvent);
352           if(signOffHTMLResponse!=null){
353             EJBAction signOffEJBHandler =
354             (EJBAction)actionMap.get(signOffHTMLResponse.getClass().getName());
355             if(signOffEJBHandler!=null){
356               EventResponse signOffEJBResponse =
357                 signOffEJBHandler.perform(signOffHTMLResponse);
358             }
359           }
360         }
361       }
362       catch(Exception e ){
363         e.printStackTrace();
364       }
365     }
366   }
```