Automated Environment Generation for Software Model Checking

Oksana Tkachuk, Matthew B. Dwyer
Department of CIS
Kansas State University, USA
oksana@cis.ksu.edu

Corina S. Păsăreanu
Kestrel Technologies
Moffett Field, USA
pcorina@email.nasa.arc.gov

Abstract

A key problem in model checking open systems is environment modeling (i.e., representing the behavior of the execution context of the system under analysis). Software systems are fundamentally open since their behavior is dependent on patterns of invocation of system components and values defined outside the system but referenced within the system. Whether reasoning about the behavior of whole programs or about program components, an abstract model of the environment can be essential in enabling sufficiently precise yet tractable verification.

In this paper, we describe an approach to generating environments of Java program fragments. This approach integrates formally specified assumptions about environment behavior with sound abstractions of environment implementations to form a model of the environment. The approach is implemented in the Bandera Environment Generator (BEG) which we describe along with our experience using BEG to reason about properties of several non-trivial concurrent Java programs.

1 Introduction

Model checking the source code of realistic software systems is a challenge and is currently the topic of a large number of research efforts (e.g., [7, 16, 30]). The primary challenge lies in overcoming the enormous cost of model checking which grows as the product of the number of independent program components, such as, threads of control. Most researchers agree that abstraction is the key to overcoming this challenge. Research on abstracting the data state of programs using techniques such as predicate abstraction (e.g., [3]) are steadily increasing the size and complexity of programs that can be efficiently analyzed. A complementary approach involves decomposing the program, checking properties of the components, and then composing the analysis results to draw conclusions about the overall behavior of the program. A variety of forms of compositional or modular verification have been studied (e.g., [18]) but they have not been adapted for software written in modern programming languages.

In this paper, we describe automated tool support for adapting existing software model checking frameworks to provide a restricted form of modular verification. Specifically, we consider decomposition of a Java program into two parts: a *unit under analysis* (henceforth called a *unit*) and its *environment*. A unit is any collection of Java classes and its environment consists of the classes with which the unit interacts through the unit's *interface*[1]. The unit's source code will be the subject of verification along with an abstract model of the environments externally observable behavior. This environment model is derived from specifications written by the user or from the results of analyzing source code that implements environment components. Existing abstraction techniques [9] may be applied to local unit data and to the data that flows between the unit and environment. The resulting abstracted unit and environment may then be analyzed by existing Java model checking frameworks such as JavaPathFinder [30] and Bandera [7].

Thorough treatment of the mechanisms by which the environment may influence the behavior of the unit is essential for sound reasoning. The environment may influence the unit's *control* (e.g., by invoking methods in the unit's interface or influencing synchronization relationships) and *data* (e.g., by passing *environment data* to the unit or by modifying *unit data* that may flow to the environment). By unit data we mean objects of *unit type* (i.e., the object's type is included in the unit). Java classes are broken into two categories depending on whether they hold a thread of control. In Java, a class containing the `main` method or classes that extend(implement) `java.lang.Thread(java.lang.Runnable)` are labeled *active*, the rest are termed *passive*. For consistency, we reuse terminology from previous work [10], and call the active environment classes *drivers* and passive environment classes *stubs*. Our approach provides mechanisms by which a wide-range of driver and stub behaviors may be safely ap-

[1] We treat interfaces in Java as classes which comprise a unit *interface* in our terminology.

proximated.

Experience has shown that the developing environment models for software model checking that are sufficiently precise to enable effective reasoning yet not so over-restrictive that they mask faulty system behaviors is a significant challenge [19, 20]. Developing such an environment may require an understanding of unstated assumptions about system usage and software interfaces, careful coding to ensure that those assumptions are satisfied in the least restrictive way, and evaluation through model checking of the environment and the unit under analysis. For this reason, we believe that *multiple sources of information should be combined to generate environment models* that reflect a broad range of realistic execution contexts for a unit under analysis. BEG is aimed at both minimizing the effort required to generate environment models and increasing their fidelity with respect to assumptions about environment behavior. Specifically, BEG currently automates: the *discovery* of the unit-environment interface based on minimal user supplied information, the synthesis of environment drivers from specifications of the sequences of program actions they may perform, and the synthesis of environment stubs from analysis of the possible program actions executed by existing environment code. Program actions in our setting are statements that may directly influence the data or control state of the unit.

We envision two ways in which environment generation tools 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 class(es) 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. They will need 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 comprise the unit under analysis and an 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.

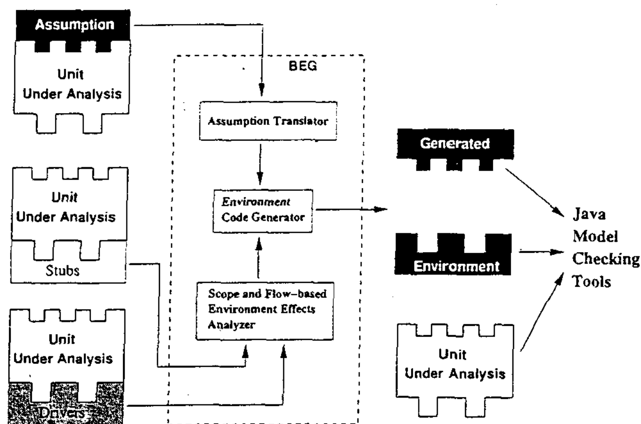

Figure 1. BEG Architecture

Our approach builds on existing work in assume-guarantee reasoning and in program flow analysis. The approach is implemented in the Bandera Environment Generator (BEG) which supports modular checking of Java source code. Our previous work [28] presented the details of the program analysis and synthesis techniques used to model the data effects of environment implementations. This paper focuses on control effects for active environment components and makes several contributions, including (*i*) defining a language of program actions with which to specify environment behaviors; (*ii*) adapting existing specification forms for defining patterns of environment behavior; (*iii*) synthesing source code models of environment behavior that can be processed by existing model checking frameworks; and (*iv*) a preliminary evaluation of the effectiveness of BEG in supporting modular source-code model checking. While BEG supports the checking of Java source code, the fundamental concepts it embodies are much more broadly applicable.

The next Section describes our basic approach and an example that is used throughout the paper. Section 3 describes the formalisms for specifying environment behavior and generating environment models as Java source code. Section 4 discusses the soundness of environments relative to specified assumptions. An overview of several case studies using BEG is presented in Section 5. We then compare and contrast our work to existing research in Section 6 and conclude in Section 7.

2 Basic Approach and An Example

The fundamental assumption in BEG is that precise reasoning about the unit is desired, but that *some* precision in modeling the environment may be sacrificed. Our approach is to *safely* approximate environment data and the environ-

ment statements that may influence the unit's behavior.

Modelling the effect of environment statements is achieved by a combination of user specifications and analysis of Java source code. Figure 1 shows the architecture of BEG. BEG accepts multiple information sources for generating an environment model. Users identify the unit under analysis by naming the classes, interfaces and packages that comprise the unit. Users provide specifications of their *assumptions* about the patterns of unit method calls and unit field definitions that the environment may make. If an implementation of the environment is available, BEG may be used to automatically extract the environment assumptions using static analysis techniques. Thus, environment models can be synthesized from a combination of assumption specifications and the results of analyzing implementations. Those models are encoded as Java source code using a collection of *modelling primitives* to express the atomic execution of environment actions, to encode non-determinism in the environment, and to reflect the approximation in analysis results.

We illustrate our approach on a small publish-subscribe program implemented using Java's `Observer` and `Observable` library components. Figure 2 shows class `Subject`, which is an observable; a field `obs` of type `Buffer`, shown on the left side of Figure 4, is a container for `Watchers` that are registered for the `Subject`. The `Watcher` class contains two bookkeeping fields that record the total number of registration `attempts` and the number of `aborts`. Suppose, we are interested in reasoning about whether "Only registered `Watchers` are notified of `Subject` updates". This can be specified in several ways, but one approach is to test whether `registered` field of `Watchers` is `true` at the point where a `Subject` calls `update()`.

2.1 Interface Discovery

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

These classes are analyzed to determine: the fields of unit supertype classes that are referenced in the unit and the non-unit classes that are directly referenced by the unit. Any referenced supertypes are included in the unit. Directly referenced non-unit classes define the *unit interface*.

For our example and the mentioned property, `Subject` and `Watcher` should be in the unit. Their supertypes `java.util.Observable` and `java.util.Observer` and referenced `Buffer` are in the environment. Note, that the actual environment may consist of more classes due to transitive class and method dependencies, however, BEG identifies classes

```
public boolean unregister(Watcher w) {
 if (super.removeElement(w)) {
  w.registered = false; return true;
 }
 return false;
}
public Watcher removeFirst() {
 Watcher result = elementAtIndex(0);
 removeElement(result);
 return result;
}
```

Buffer Implementation

```
public boolean unregister(Watcher p0){
 if(chooseBool()) p0.registered = false;
 return chooseBool();
}
public Watcher removeFirst() {
 return ((Watcher)chooseClass("Watcher"));
}
```

Generated Environment

Figure 4. Bounded Buffer Stubs (excerpts)

that are immediately referenced in the unit. The part of the environment that is invisible to the unit is safely approximated.

2.2 Driver Specification and Synthesis

One may specify assumptions about sequences of method calls and unit field definitions that the environment may make on the unit. BEG generates a set of *driver threads* that implement the most liberal model that is consistent with the given assumptions. Figure 3 illustrates an assumption with one instance of `Subject` and two `Watchers` and a pair of threads whose behavior is given by regular expressions over method names with parameter values elided; elided parameters means that their value is selected non-deterministically from the possible values of the parameter type. The first thread repeatedly calls the `changeState()` method on a selected `Subject` and the second calls any sequence of `add()` or `delete()` calls on a selected `Subject` with *some* `Watcher`.

Figure 3 also shows the generated drivers that capture the assumed behavior. `Main` allocates the specified instances and starts the execution of the two threads. Thread implementations model the assumption specifications by invoking *modeling primitives* that capture non-determinism (e.g., `chooseBool()` chooses among $\{true, false\}$, `chooseInt(n)` chooses among $\{0, \ldots, n\}$, and `chooseClass(C)` chooses among the allocated instances of class `C`) [8].

2.3 Stub Analysis and Synthesis

A series of static analyses, including points-to and side-effects analyses, are applied to determine how the environ-

```
public class Watcher implements Observer{
  static public int attempts = 0;
  static public int aborts = 0;
  public boolean registered = false;
  public void update(Observable o, Object arg) { }
}
public class Subject extends Observable {
 boolean changed = false;
 Buffer obs;
 public Subject() { obs = new Buffer(); }
 public void changeState() { setChanged(); notifyObservers();} }
 public synch... void add(Watcher o) { obs.register(o);}
 public synch... void delete(Watcher o) { obs.unregister(o);} }
```

```
public void notify(Object arg) {
 Watcher cw;
 Buffer lb = new Buffer();
 synchronized (this) {
  if (!changed) return;
  obs.copy(lb); changed = false;
 }
 if (obs.size() != lb.size()) cw = null;
 while (!lb.isEmpty()) {
  cw = lb.removeFirst(); cw.update(this, arg);}
protected synch... void setChanged() { changed = true;}
```

Figure 2. Customized Observer Implementation

```
environment {
 instantiations { 1 Subject; 2 Watcher; }
 regular assumptions {
   (changeState())*;
   (add() | delete())*;
  }
}

public class EnvDriver {
  public static void main(java.lang.String[] param0){
    Subject s0 = new Subject();
    Watcher w0 = new Watcher();
    Watcher w1 = new Watcher();
    new T0(s0, w0, w1).start();
    new T1(s0, w0, w1).start();
  }
}
```

```
public class T0 extends java.lang.Thread {
 public Subject s0;
 public Watcher w0, w1;
 public T0(Subject p0, Watcher p1, Watcher p2){
  s0 = p0; w0 = p1; w1 = p2;}
 public void run(){
  while(Bandera.chooseBool()) s0.changeState();}
}
public class T1 extends java.lang.Thread { ...
 public void run(){
  while(Bandera.chooseBool())
   switch(Bandera.chooseInt(2)){
   case 0: s0.delete(Bandera.chooseClass("Watcher"));
    break;
   case 1: s0.add(Bandera.chooseClass("Watcher"));
    break;}}
}
```

Figure 3. Assumptions and Generated Drivers

ment methods can influence the unit data [28]. In our example, the analysis of the `Buffer` implementation calculates effects of the environment on the fields of `Watcher`, for instance method `unregister` may side-effect only one field `registered`.

Models are generated to reflect all possible side-effects as calculated by the preceding analyses. To safely reflect the *possibility* of a side-effect, code is generated to execute *abstract assignments* non-deterministically. Figure 4 shows the generated environment for several methods of `Buffer`. For example, the access of a `Watcher` instance, via call `elementAtIndex(0)`, in method `unregister()` is approximated as the return of a non-deterministically chosen instance of `Watcher`.

2.3.1 Tool Support

This example illustrated the basic capabilities of BEG. BEG supports the specification a wide-range of assumptions about environment behavior compactly using regular expressions, temporal logic formula (defined over program actions), and data side-effects summaries. In the absence of specified assumptions, BEG can be configured to make *reasonable* assumptions about the intended environment. For example, it is assumed that the calling environment consists of a number of unit class instances and threads (specified on the command line) that exhibit *universal* behavior (i.e., they perform any sequence of calls over the methods in the system by selecting appropriately typed class instances).

In its current form, the tools make assumptions about the lack of divergence indefinite-blocking, and lock acquisition in the environment. Ongoing work is extending the tools to support the specification of behavior related to these language aspects and the extraction of safe approximations of such behavior from implementations. Despite these limitations, the BEG toolset has been effective in supporting modular reasoning about properties of a number of realistic systems as discussed in Section 5.

3 Driver Specification and Synthesis

We focus in this section on the specification of the expected behavior of environment drivers. The building blocks of those specifications are descriptions of program actions that may influence the control or data state of the unit under analysis. Those program actions are then combined to describe the patterns of environment behavior.

3.1 Environment Instantiation

We define a *name scope* within which environment specifications may refer to specific class instances; by default the name scope is empty.

The *global* name scope is defined by annotating instantiations. In an instantiation, the number of instances allocated of a type by the environment is given and those instances may be named. For example, we can adapt the assumption specification in Figure 3 to explicitly name the lone Subject instance, s, and reference it regular expression.

```
environment {
 instantiations { 1 Subject s; ... }
 regular assumptions { (s.add() | s.delete())*; ... }
}
```

It is important to distinguish between *named* instances and the set of all instances. The latter is the set of all environment and unit allocated instances. That set forms the universe from which non-deterministic choice primitives over reference types are evaluated.

A *local* name scope can also be defined that applies to a portion of an assumption specification. The preceding example can be rewritten using a local name scope as:

```
environment {
 instantiations { 1 Subject; ... }
 regular assumptions {
  <Subject s>:(s.add() | s.delete())*; ... }
}
```

By default local names are bound to a non-deterministically selected value of the given type that holds throughout the name scope (which is denoted explicitly by a pair of { } and which extends to the end of the expression by default). Thus, they serve a function similar to universal quantifiers in logics and their primary use is in correlating event occurrences (e.g., that a sequence of actions are applied to the *same* receiver object).

In these examples, there is a single instance of Subject, thus the three specifications are semantically equivalent. In general, this will not be the case. Local name introduction is interpreted as non-deterministic choice over the the set of allocated instances of the named type, Subject in this case. Local name scopes may be nested and may refer to additional names. For example, <Subject x>:<Subject-x y>:... introduces two names that are guaranteed to refer to distinct instances of Subject. Local names may also be bound to values from the unit. For example, <Ref x=getRef()>:x.m() introduces a name x that is bound to the value returned by a call and that is subsequently used to perform a call on method m().

3.2 An Alphabet of Program Actions

Let U be the set of classes that comprise the unit under analysis and let B denote the set of Java builtin types. We define an alphabet of actions consisting two classes of actions: field assignments and method calls.

Assignments can be either static field assignments or assignments through object references of unit type. Assignments are of the form $r.f = rhs$ where: $type(r) \in U, f$ is of unit type, and *rhs* is either a scalar constant or $\top_{type(f)}$, if $type(f) \in B$, or chooseClass(*type(f)*), if $type(f) \in U$. Here $\top_{type(f)}$ denotes any possible value of *type(f)*; for scalar types the expansion of values is done implicitly via abstraction [9]. The target of the assignment, r, is either an introduced name, $\top$ for an appropriate type, or the name of a class.

Method call actions are defined using standard Java syntax, but where partial specification of parameters is allowed. Consider a method in class C with signature public R m(P1 p1, P2 p2). We can denote the occurrence of a call to this method with any receiver object of type C, a specific value, v_1, for p_1, and any value for p_2 as $m(v_1, \top)$, where v_1 is an introduced name or a scalar. Partially specified calls may omit the receiver object or any parameter by replacing it with $\top$. The meaning of such a call is the set of all calls that can be constructed by replacing $\top$ with any legal value of the receiver or parameter type.

We note that BEG, through the process of interface discovery, produces the set of program actions (i.e., public method calls and fields at the unit-environment interface) that can be used to define assumptions.

3.3 Specifying Patterns of Actions

Regular expressions defined over this alphabet 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: $r; s$ (concatenation), $r|s$ (disjunction), $r*$ (closure), and $r?$ (one or more occurrences of r). Positive closure ($r+$), bounded iteration ($r * n = r_1; r_2; \ldots; r_n$), and a generalization of bounded iteration ($r * \{n, m\} = r * n | r * (n+1) | \ldots | r * m$) are also supported. These expressions can appear in introduced name scopes, where those names are referenced in the program actions used in the expression. The syntax of these assumptions is given in Figure **??**. where a is a program action, a_{fun} are *function call* actions, n and m are introduced name, and t is a program type name. Legal assumption specifications must also satisfy some type constraints. Specifically, type expressions, *te*, may only involve types and named variables, m, of that type; m here must refer to a name introduced in an enclosing name scope. Similarly, name initializations, *ni*, may only involve function call actions whose type is compatible with the type of the type expression for the introduced name.

As an example, java.util.Iterator presents a simple 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

r	$:==$	a
	$:==$	$.$
	$:==$	$[a_1, a_2, \ldots, a_n]$
	$:==$	$[-a_1, a_2, \ldots, a_n]$
	$:==$	$s_1 ; s_2$
	$:==$	$s_1 \vert s_2$
	$:==$	(s)
	$:==$	$s?$
	$:==$	$s*$
	$:==$	$s+$
	$:==$	$s + i, j$
	$:==$	$< t\ ni >: s$
	$:==$	$< te\ n >: s$
ni	$:==$	n
	$:==$	$n = a_{fun}$
te	$:==$	t
	$:==$	$te - m$

Figure 5. Assumption Syntax

with the following specification:

```
environment {
 instantiations { k Iterator; }
 regular assumptions {
   [Iterator i]:i.iterator();
                (i.hasNext(); i.next(); i.remove()?)*
 }
}
```

This expresses both required sequencing of calls (e.g., a call to `iterator()` 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`.

3.4 From Regular Expressions to Code

Java models of regular expresson assumption specifications can be generated using the templates shown in Figure 6. 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.

One can view name scope introduction for a subexpression as prefixing a special *name binding* action to the subexpression. Name scopes are supported by introducing local variables in the body of the driver `run()` method and assignments that non-deterministically choose an instance to be bound to the name at the point where the name binding action is embedded in the regular expression.

We note that much of the generated model code is internal to the environment. Internal environment states and actions are *hidden* in our models by embedding them in atomic statements. This 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.

```
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+                    →  do { code(r);} while (chooseBool()
r?                    →  if (chooseBool()) {code(r);}
r*n                   →  for (int i=0;i<n;i++) {
                           code(r);}
                         }
r*{n,m}               →  for (int i=0;
                              i<n1+chooseInt(m-n);i++) {
                           code(r);
                         }
<t n>:r               →  { t n = chooseClass(t);
                           code(r);}
<t n=f>:r             →  {t n = f();
                           code(r);}
<t-m1-...-mk n>:r     →  { t n;
                           while (true) {
                             n = chooseClass(t);
                             if (n == m1) continue;
                             ...
                             if (n == mk) continue;
                             break;
                           }
                           code(r);}
                         }
```

Figure 6. Assumption Semantics

Regular expressions are a familiar formal notation to many developers and our experience is that many find it easier to use than temporal logics. We also support assumptions specified as Linear Temporal Logic (LTL) and generate Java models using an approach that is similar to the one developed for Ada modeling in [22].

4 Soundness of Synthesized Environments

In this section, we justify the soundness of synthesized environments with respect to assumption specifications and the results of side-effects analyses.

4.1 Preliminaries

Formally, we model the behavior of a concurrent program written in Java as a *labelled transition system.* Corbett shows how to model the behavior of Java [6] programs as transition systems as defined below, using standard techniques for constructing control flow graphs.

A *labelled transition system* P is a triple $\langle S(V), \texttt{Act}, R\rangle$, where V is a set of *typed program variables*, $S(V)$ is the set of states representing valuations of the variables from V, Act is an alphabet of actions and $R \subseteq S(V) \times \texttt{Act} \times S(V)$ is a transition relation. We write

$s \overset{a}{\longmapsto} s'$ for $(s, a, s') \in R$. For a set of variables $W \subseteq V$, $s|_W$ denotes the valuation of variables from W in state s.

States of a system can be regarded as tuples giving the values of all relevant program variables, including the program counter. A transition $s \overset{a}{\longmapsto} s'$ says that the system can evolve from state s to state s' by executing action a. The labels on transitions can represent variable assignments, variable tests, and actions modelling transfer of control to and from a procedure; parameter passing can be simulated by communication through common (shared) variables.

We assume that a system is *open*, i.e. it can interact with its environment through shared variables and actions. If V is the set of variables for an open system, let V^{int} denote the set of *internal (local) variables* (that only the system itself may modify) and let V^{com} denote the set of *common (shared) variables*, such that $V = V^{int} \cup V^{com}$ and $V^{int} \cap V^{com} = \emptyset$. Also if $\texttt{Act}$ is the set of actions of the open system, let $\texttt{Act}^{int}$ denote the set of *internal actions* (a symbol representing an internal action of a system is in the alphabet of only that system) and let $\texttt{Act}^{com}$ denote the set of *communication (or interface) actions*, such that $\texttt{Act} = \texttt{Act}^{int} \cup \texttt{Act}^{com}$ and $\texttt{Act}^{int} \cap \texttt{Act}^{com} = \emptyset$. A system is *closed* if it may not interact with the environment; a closed system has no shared variables or actions (i.e. $V^{int} = \emptyset$ and $\texttt{Act}^{int} = \emptyset$).

In order to define the interaction of an open system with the environment, we first define a parallel composition of two systems. Let $P_1 = \langle S(V_1), \texttt{Act}_1, R_1 \rangle$ and $P_2 = \langle S(V_2), \texttt{Act}_2, R_2 \rangle$ be two open systems. We say that P_1 and P_2 are *compatible* if both their sets of internal variables and sets of internal actions are disjoint (i.e. $V_1^{int} \cap V_2^{int} = \emptyset$ and $\texttt{Act}_1^{int} \cap \texttt{Act}_2^{int} = \emptyset$).

Let P_1 and P_2 be two compatible systems as above. The *composition* of P_1 and P_2, denoted $P_1||P_2$, is another system $P = \langle S(V), \texttt{Act}, R \rangle$, where $V = V_1 \cup V_2$, $\texttt{Act} = \texttt{Act}_1 \cup \texttt{Act}_2$, $(s, a, s') \in R$ iff $(a \notin \texttt{Act}_i \wedge s|_{V_i^{int}} = s'|_{V_i^{int}}) \vee (a \in \texttt{Act}_i \wedge (s|_{V_i}, a, s'|_{V_i}) \in R_i)$, $i = 1, 2$.

The two systems synchronize on the shared actions and asynchronously interleave all other actions. The internal variables of system P_i may be modified only by the actions of system P_i, while the common variables may be modified by both systems.

An *environment* for system P is another system E that is compatible with P. Note that after completing the system with a definition of a system representing the environment, the resulting system (i.e. $E||P$) is still *open*, admitting arbitrary interference from the environment; once we know that all the processes/code modelling the environment have been included, and no further interaction with the external world is expected, we may "close" system $E||P$ by declaring all the shared variables and actions to be internal to the system.

4.2 Data and Control Effects

Our program model is general enough to capture different interactions between the system and the environment: through *shared data* and *control* (i.e., *communication actions*); the model does not directly capture dynamic allocation of data, so we put a limit to the number of objects that can flow into the system from the environment.

Our generated environments are either *drivers* or *stubs*. Drivers capture the control influences from the environment, while the stubs capture the data influences from the environment.

4.2.1 Simulation and Preservation Results

We proceed to define when a system is a *sound abstraction* of another one. Abstracting means having less details while respecting behaviors of the original system. Let $P = \langle S(V), \texttt{Act}, R \rangle$ and $P' = \langle S(V'), \texttt{Act}', R' \rangle$ be two systems. We say that P is a *sound abstraction* of P' iff there is a simulation from P to P'.

A *simulation* [18] from P to P' is a pair (ρ_s, ρ_a) of relations with $\rho_s \subseteq S(V) \times S(V')$ and $\rho_a \subseteq \texttt{Act} \times \texttt{Act}'$ such that if $(s, s') \in \rho_s$ and $s \overset{a}{\longmapsto} t$, then there exists some state $t' \in S'$ and some action $a' \in \texttt{Act}'$ such that $s' \overset{a'}{\longmapsto} t'$, $(t, t') \in \rho_s$ and $(a, a') \in \rho_a$. We say that P *simulates* P', denoted $P \preceq P'$, if there is a simulation from P to P'.

When specifying properties of software systems, we use *universal temporal logics*, i.e., we reason about properties that hold along every possible execution path. A standard result, see e.g. [21], says that simulations preserve satisfaction of formulas of such logics. I.e., if $P \preceq P'$, then, for every universal temporal formula ϕ, $P' \models \phi$ implies $P \models \phi$. However, if $P' \models \phi$ does not hold, it does not mean that $P \models \phi$ is necessarily false (i.e., completeness is sacrificed).

Our synthesized environments are *sound abstractions* of real environments (see [21, 26]), and model checking a system in a synthesized environment is sound. Extending of the results from [21, 26], we have the following results: (i) if $E \preceq E_{abs}$, then $E||P \preceq E_{abs}||P$, and (ii) if $E \preceq E_{abs}$ and $P \preceq P_{abs}$, then $E||P \preceq E_{abs}||P_{abs}$. In other words, it is safe to check universal temporal properties in the programs that use the automatically generated environments since these environments are sound abstractions of real environments.

5 Experience with BEG

BEG is implemented using the SOOT framework [29]. BEG uses SOOT's symbol table, control flow graph, and bytecode representation, Jimple, to perform its analyses; Jimple is a 3-address SSA-like intermediate form. The tools

produce Java code as output that includes calls to the modeling methods introduced in Section sec:GENERATE. This section describes our experience applying BEG to generate environments for portions of programs that have appeared recently in the literature on Java verification. The actual verification was performed using either JavaPathFinder or Bandera.

5.1 Case Studies in Environment Generation

We have applied BEG to a variety of examples[2]. 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; the resulting checks are in fact slightly more efficient due to the atomicity of environment behavior in generated environments. In addition to the Observer/Observable example, these examples include: a **Producers/Consumers** framework for exercising a bounded buffer [15]; **RWVSN**, Lea's [17] generic readers-writers synchronization framework; and dining philosophers with host, a classic synchronization problem.

While BEG proved to be quite 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. 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 is a Java framework that supports the development of graphic user interfaces (GUI)). Frameworks present rich interfaces that allow application specific processing to be co-ordinated through the framework. Frameworks are quite 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. BEG enables the synthesize of drivers that capture multiple framework use cases and mode state machines. Furthermore, the use of non-determinism in assumption specifications allows drivers to span configuration settings. This has the great advantage of allowing configuration-independent properties to be analyzed without having to enumerate combinations of configuration settings.

To explore BEG's support for analyzing frameworks, we consider two non-trivial Java programs. **Autopilot** is a swing-based GUI for an MD-11 autopilot simulator used for pilot training at NASA [24]; it is a framework client application. **ReplicatedWorkers** [12], is a parameterizable parallel job scheduling framework. Since neither of these programs can be model checked efficiently in combination with an environment implementation, rather than focus on measures of time and space required for checking, we describe how the tools supported the user in performing modular checking.

[2]The details of all examples are given at `http://www.cis.ksu.edu/bandera`.

5.2 Autopilot

The MD-11 autopilot tutor is a web-based application that has a graphical user interface (GUI) that simulates the Autopilot Mode Control Panel and a Primary Flight Display of an MD-11 aircraft autopilot. A user may click on buttons to dial desired altitude and vertical speed, and advance the aircraft towards its goal altitude. Autopilot is implemented as an applet. The application code consists of more than 3600 lines of code clustered in two main classes. 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 system was checked for *mode confusion* by encoding a model of a pilot's understanding of the aircraft state. That *mental model* was integrated with the system to monitor GUI inputs. Assertions were inserted to compare the state of system data structures with the state of that model; assertion violations indicated a mismatch between the mental model and the software's state which implies a potential mode confusion.

To analyze the system BEG was used to generate stubs for all the GUI framework components and to generate drivers that encode regular assumptions about pilot behavior. We restrict our attention here to the generation of the drivers, for a more complete description see [27].

The main class of the system is `Autopilot` which extends `java.applet.Applet` which in turn extends several `AWT` classes. This applet makes a large number of calls to `AWT` methods in order to create and update the simulated cockpit displays. The properties we wished to reason about, however, were independent of the state of the GUI and we were chose the `Autopilot` class itself as the unit.

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.

For this system, we found it useful to name the pilot actions to improve the readability of both the assumption specifications and generated counter-examples. As shown in Figure 7, BEG allows one to define mnemonics for GUI interface actions and to define regular assumptions in terms of those mnemonics. Model checking the `Autopilot`

```
environment{
 instantiations { 1 User(new Autopilot()); }
 definitions {
  start=mouseClicked(1); pullAltKnob=mouseClicked(6);
  incMCPAlt=mouseClicked(9); incMCPVS=mouseClicked(11);
  fly=mouseClicked(14); pilotExp=getExpectation();
 }
 regular assumptions {
  start > incMCPAlt^{1,10} >
  pullAltKnob > (pilotExp > fly)^{1,10} >
  incMCPVS^{1,10} > (pilotExp > fly)^5 ;
 }
}
```

Figure 7. `Autopilot` **Assumptions**

```
environment {
 import ca.replicatedworkers.*;
 instantiations {
  1 ConcreteWorkCollection; 1 ConcreteWorkItem;
  1 ConcreteResultsCollection; 1 ConcreteResultItem;
  1 ReplicatedWorkers(
      new Configuration(NONE, SYNCH, SOME),
      TOP, TOP, 2, 1, 1, 0);
 }
 regular assumptions {
  (putWork(TOP) > execute())* > destroy();
 }
}
```

Figure 8. `RW` **Assumptions**

class with the generated environment using JavaPathFinder produced the following counter-example:

```
start > incMCPALT^2 > pullAltKnob > fly^2 > incMCPVS > fly
```

which indicated a mode confusion anomaly that is possible in the tutor.

It is interesting to note, that a previous effort to build an environment for this application required approximately 6 months and yielded an environment model that was inconsistent with the actual environment implementation. From relatively simple specifications, BEG generated an environment in less than 4 minutes that is guaranteed to be consistent with the implementation, modulo the fidelity of assumption specifications.

5.2.1 Replicated Workers

`Replicated Workers` (RW) is a configurable framework designed to support the parallelization of simulations. In previous work [10], we applied largely manual techniques to model check a collection of properties of an Ada implementation of this framework. Subsequent to that work the framework was rewritten in Java and has been widely used [12].

Like most frameworks, replicated workers instances create threads internally. Clients control the degree and asynchrony of parallelism in the configuration by passing parameters to the constructor of the framework instance. The replicated workers framework makes significant use of interfaces to enable call-backs to the client supplied computations that are to be parallelized. An environment for the replicated workers, must define and instantiate classes that implement each of the interfaces given in the framework and define appropriate configuration information.

We checked several properties from [10] and were able to reproduce those results with one difference. When checked a framework instance under the environment defined in Figure 8 for deadlock, we found an actual deadlock. The bug was in the Java implementation of a barrier synchronization utility. Its discovery was surprising since the framework has been used in implementing more than ten non-trivial parallel simulation applications and this bug was never discovered. We replaced the barrier implementation with one from `java.util.concurrent` and the deadlock was eliminated.

6 Related Work

Modular approaches to model checking have been studied for more than a decade. This work has been carried out mostly at the theoretical level although there have been some implementations of game-theoretic approaches to reasoning about open systems (e.g., [1]). Our focus is on capturing the complexities of unit/environment interaction that arise in real programming language and supporting the specification and extraction of precise, yet compact environment models.

Environment generation from specifications presented in this paper builds on work of Avrunin et al. [2], who developed tool support for analyzing partially implemented real-time systems with some components implemented in Ada and others described using graphical interval logic and regular expressions, and our previous work on model checking of partial software systems in Ada [10, 11, 22]. In addition to treating Java programs, we support a much richer class of environment specifications and extract environment models from existing code.

Another modular approach to checking multithreaded programs is implemented in Calvin [13]. Their approach is aimed at procedure checking relying on a user specifications of environment 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 in interacting.

As complementary to our approach, generation of environment assumptions for *optimistic* environments has been described in [14, 4] Their work is aimed at finding environments within which the unit would satisfy its required properties. This is an important direction to pursue for modular program checking, but we also believe that extraction

of environments plays an important role when using model checking as a kind of *unit testing* approach on existing code bases.

There is a number of examples of applying static analysis techniques used in modular analysis or verification. Verisoft incorporates a static analysis to closing of open systems by calculating the influence of externally defined data [5]. Unlike in our approach, they use a simple notion of data dependence to drive their analysis and do not have the ability to control the precision of the generated system. Stoller [25] 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. BEG generates environment values based on the user specification or the assumption that the environment data is to be abstracted before model checking phase. Rountev et al. [23] 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 our work, their summries are produced using whole program analysis under the worst-case assumptions about a client and are targeted at the optimizations of the client. Our analyses are modular and explore the information about the unit, if there are call backs from the environment.

7 Conclusions

Despite the significant computational complexity of model checking, it has proven effective as an analysis technique that is capable of finding errors in real concurrent Java programs (e.g., the Replicated Workers framework). Modular approaches promise to further scale the application of model checking to software. The Bandera Environment Generator (BEG) provides automated tool support that has proven effective in enabling useful forms of modular analysis.

We are continuing development of the foundations for BEG as well as the tool support. Specifically, we are working on analysis of program lock acquisition to safely approximate the synchronization interaction between the environment and unit. In addition, we are adapting thread modular approaches [13] to enable model checking for arbitrary numbers of environment threads. BEG is being released as part of the Bandera toolset at `http://www.cis.ksu.edu/bandera`.

8 Acknowledgments

The authors would like to thank the members of the Bandera and JPF projects for many helpful discussions and comments related to this work. This work was supported in part by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement DAAD190110564, by DARPA/IXO's PCES program through AFRL Contract F33615-00-C-3044, and by Intel Corporation under grant 11462.

References

[1] R. Alur, L. de Alfaro, R. Grosu, T. A. Henzinger, M. Kang, C. M. Kirsch, R. Majumdar, F. Mang, and B.-Y. Wang. jmocha: A model-checking tool that exploits design structure. In *Proceedings of the 23rd Annual IEEE/ACM International Conference on Software Engineering*, pages pp. 835–836. IEEE Computer Society Press, 2001.

[2] G. S. Avrunin, J. C. Corbett, and L. Dillon. Analyzing partially-implemented real-time systems. In *Proceedings of the 19th International Conference on Software Engineering*, pages 228–238, 1997.

[3] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 203–213. ACMPress, June 20–22 2001.

[4] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference (LNCS 2619)*, Apr. 2003.

[5] 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.

[6] J. C. Corbett. Constructing compact models of concurrent Java programs. In M. Young, editor, *Proceedings of the 1998 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, March 1998.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] M. B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency - Practice and Experience*, 9(11):1293–1310, 1997.

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

[14] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE Conference on Automated Software Engineering*, May 2002.

[15] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[17] D. Lea. *Concurrent Programming in Java[tm], Second Edition: Design principles and Patterns*. The Java Series. Addison-Wesley, 2nd edition, 1999.

[18] R. Milner. *Communication and Concurrecy*. Prentice Hall, 1989.

[19] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the deos scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[20] C. S. Păsăreanu. Deos kernel: Environment modeling using ltl assumptions. Technical Report Technical Report NASA-ARC-IC-2000-196, NASA Ames, 2000.

[21] C. S. Păsăreanu. *Abstraction and Modular Reasoning for the Verification of Software*. PhD thesis, Kansas State University, 2001.

[22] 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.

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.