Automated Environment Generation for Software Model Checking

Oksana Tkachuk, Matthew B. Dwyer
Department of CIS
Kansas State University
Manhattan, KS 66506, USA
{oksana,dwyer}@cis.ksu.edu

Corina S. Păsăreanu
Kestrel Technology LLC
NASA Ames Research Center
Moffett Field, CA 94035, USA
pcorina@email.arc.nasa.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., [8, 15, 28]). 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., [4]) 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. In order to model check a component in isolation, one needs to incorporate a model of the environment that the component interacts with. It is often the case that components satisfy properties only in specific contexts. This has given rise to the assume-guarantee style of reasoning, where the model of the environment is restricted by assumptions provided by the developer. 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 present the Bandera Environment Generator (BEG), a toolset that automates the generation of environments to provide a restricted form of modular model checking of Java programs. Specifically, we consider decomposition of a Java program into two parts: a *unit* under analysis and its *environment*. A unit is any collection of Java classes and its environment consists of the classes with which the unit interacts. The unit's source code is the subject of verification along with an abstract model of the environment's externally observable behavior. The environment model is derived from specifications written by the user, called *environment assumptions*, or from the results of analyzing source code that implements environment components. Existing abstraction techniques [10] 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 against unit properties by existing Java model checking frameworks such as Bandera [8] and Java PathFinder (JPF) [28].

Experience has shown that developing environment models for software model checking that are sufficiently precise to enable effective reasoning yet not so overrestrictive 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. BEG is aimed at both reducing the effort required to generate environment

models and increasing their fidelity with respect to assumptions about environment behavior. Specifically, BEG automates the *discovery* of the unit-environment interface based on user supplied information about the unit, synthesis of environment models from user supplied environment assumptions, and automatic extraction of environment models from environment implementations.

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) and *data* (e.g., by modifying the unit data that flows into the environment). 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 both control and data interaction between the unit and the environment. BEG allows users to specify environment properties using several formal notations. Linear Temporal Logic (LTL) [17] and regular expression can be used to capture environment assumptions as patterns of *program actions*, unit method calls or field assignments, that the environment may execute. Specifications of the possible side-effects that the execution of environment methods may produce on unit data may also be fed to BEG [24], however, in this paper we focus primarily on assumptions that capture ordering relationships among program actions.

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 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. 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

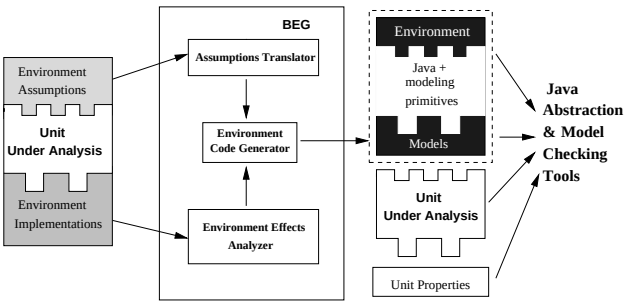

Figure 1. BEG Architecture

of the external environment.

Our approach builds on existing work in assume-guarantee reasoning and in program flow analysis. Our previous work [26] 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 of the environment 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*) synthesizing 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 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 environment statements that may influence the unit's behavior.

Modeling the effect of environment statements is achieved by a combination of user specifications and analysis of Java source code. Figure 1 shows the high-level architecture of BEG. Users identify the unit under analysis by naming the classes, interfaces, and packages that comprise the unit. BEG accepts multiple information sources

for generating an environment model. Users may 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 and then generate environment models from the extracted assumptions. Thus, environment models can be synthesized from a combination of assumption specifications and the results of analyzing implementations. Environment models are encoded as Java source code using a collection of *modeling primitives* to express the atomic execution of environment actions, to encode non-determinism in the environment, and to reflect the approximation in analysis results. The unit is combined with its environment, existing Java abstraction and model checking frameworks [8, 28] may be used to verify the unit properties.

We distinguish between two categories of Java classes depending on whether they hold a thread of control. A class containing the `main` method or classes that extend(implement) `java.lang.Thread(Runnable)` are labeled *active*, the rest are termed *passive*. For consistency, we reuse terminology from previous work [11], 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 approximated.

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 right side of Figure 2) is a container for `Watcher`s that are registered for the `Subject`. The `Watcher` class contains a bookkeeping field `registered` that keeps track of whether the `Watcher` is registered on a `Subject`. Suppose, we are interested in reasoning about whether "Only registered `Watchers` are notified of `Subject` updates". This property can be specified by asserting that the `registered` field of `Watcher`s 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. For our example and the mentioned property, `Subject` and `Watcher` should be in the unit.

Unit classes are analyzed to discover the public methods and fields of the unit that may be referenced by the environment. These are used to define the program actions that may be used to specify patterns of environment behavior. For example, public methods `changeState()`, `add()`, and `delete()` of class `Subject` may be invoked by environment components.

Unit classes are also 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 only generates environment components that are immediately referenced in the unit. BEG safely approximates the rest of the environment by incorporating summaries of classes that have an indirect effect on the unit into the generated environment components.

2.2 Driver Specification and Synthesis

One may specify assumptions about sequences of environment initiated program 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 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; elided parameters are interpreted as a value that 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 non-deterministically selected `Subject` with a non-deterministically selected `Watcher`.

Figure 3 also shows the generated drivers that capture the assumed behavior. `EnvDriver` 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 non-deterministic choice over a set of values (e.g., `chooseBool()` - a choice between $\{true, false\}$, `chooseInt(n)` - a choice over $\{0, \ldots, n-1\}$, and `chooseClass("C")` - a choice over the allocated instances of class `C` in the program state where the call is executed) [9].

```
public class Subject extends Observable {
 boolean changed = false;
 Buffer obs;
 public Subject() { obs = new Buffer(); }
 public void changeState() {
   setChanged(); notifyObservers();}
 public synchronized void add(Watcher o) {
  obs.register(o);}
 public synchronized 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 synchronized void setChanged() {
  changed = true;}
}
```

```
public class  Watcher implements Observer{
  public boolean registered = false;
  public void update(Observable o, Object arg) { }
}

public class Buffer extends Vector {
 public boolean register(Watcher w) {
  if (!contains(w)) {
   w.registered = true;
   super.addElement(w);
   return true;
  }else return false;
 }
 public boolean unregister(Watcher w) {
  if (super.removeElement(w)) {
   w.registered = false;
   return true;
  } else return false;
 }
 public Watcher removeFirst() {
  Watcher result = elementAtIndex(0);
  removeElement(result);
  return result;
 }
}
```

Figure 2. Customized Observer Implementation

```
environment {
  instantiations { 1 Subject; 2 Watcher; }
  regular assumptions {
      T0: (changeState())*
      T1: (add() | delete())*
  }
}

public class EnvDriver {
  public static void main(java.lang.String[] p0){
    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. User Assumptions and Driver Models

2.3 Stub Analysis and Synthesis

A series of static analyses, including points-to and side-effects analyses, are applied to determine how the environment methods can influence the unit data [26]. In our example, the analysis of the `Buffer` implementation calculates effects of the environment on the fields of `Watcher` (e.g., method `unregister` may side-effect the field `registered`).

Models are generated to reflect all possible side-effects as calculated by the analyses. To safely reflect the *possibility* of a side-effect, code is generated to execute *abstract assignments* non-deterministically. Figure 4 (left) shows the extracted assumptions and (right) 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` via the call to `chooseClass("Watcher")`.

2.4 Model Checking

After the environment models are generated, we combine them with the code of the unit and use JPF or Bandera to verify the unit properties. Model checking the observer example from Figure 2 with JPF using the environment models from Figures 3 and 4 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 `removeFirst()` method which can return any allocated instance of type `Watcher`. Boosting the precision of the `Buffer` model, by using a more precise analysis [26], eliminates the spurious counter-example and reveals a property violation due to race condition in the implementation of `notify()` that is due to the intentional limitation of the scope of the `synchronized` statement for improved performance.

```
method register(Watcher p0):
 p0.registered = true
 return chooseBool();

method unregister(Watcher p0):
 p0.registered = false
 return chooseBool();

method removeFirst():
 return chooseClass("Watcher")
```

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

Figure 4. Buffer Extracted Assumptions and Stub Model

2.5 Tool Support

The observer example illustrated the basic capabilities of BEG. BEG supports specification of a wide-range of assumptions about environment behavior compactly using regular expressions, LTL (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 (specified on the command line) of unit class instances and threads that exhibit *universal* behavior (i.e., they perform any sequence of program actions that are exposed at the unit-environment interface). In addition, BEG can be used to extract environment assumptions from environment implementations [26]. With these capabilities, 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

In this section, we focus on the specification of the expected behavior of environment drivers. The building blocks of those specifications are definition of data instances that the environment may reference, description of the program actions over those instances that may influence the control or data state of the unit under analysis, and the specification of patterns of actions that correspond to assumed environment behavior. We describe each of these in turn in the remainder of this section.

3.1 Environment Instantiation

A *name scope* defines a region within an environment specification in which the developer may introduce names that are bound to specific class instances.

There is a single *global* name scope that is defined by annotating environment instantiations with introduced names. In an instantiation, the number of instances allocated of a type by the environment is given and those instances may be named. The bindings of names to values holds throughout all of the specified assumptions.

Named instances are a subset of the set of all instances of a type. The latter being 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.

For example, we can rewrite the assumption specification in Figure 3 to explicitly name the lone `Subject` instance, `s`, and reference it in the regular expression as:

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

A *local* name scope defines a value for an introduced name that applies to only a portion of an assumption specification. For clarity in environment specification, the reuse of names between global and local scopes is disallowed. By default local names are bound to a non-deterministically selected value of the given type; that binding holds throughout the body of the expression that follows the name introduction. Thus, local names serve a function similar to universal quantifiers in logics and their primary use is in correlating the occurrences of different program actions (e.g., to specify that multiple actions are applied to the *same* receiver object).

The preceding example can be rewritten using a local name scope as:

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

In these examples, there is a single instance of `Subject`, thus the three specifications discussed above are semantically equivalent; in general, this will not be the case. Local name scopes may be nested and they may restrict the universe from which values are non-deterministically selected. For example, `<Subject x>:<Subject-x y>:...` introduces two names, `x` and `y`, 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()`. The syntax of name scope is given in Figure 5 which is presented in detail at the end of this section.

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 built-in types. We define an alphabet of actions consisting of 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 scalar or unit type, and *rhs* is either a scalar constant or TOP, which denotes any possible value of *type*(*f*). The target of the assignment, r, is an introduced name, `chooseClass("C")` for a desired type C, 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 the value TOP for the formal parameter type. 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 any value for `y` as `m(`p_1`,` TOP`)`. The meaning of such an action is the non-deterministic 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 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 legal parameter values.

3.3 Specifying Patterns of Actions

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 ($r + \{n, m\} = r + \{n\} | r + \{n+1\} | \ldots | r + \{m\}$) 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 assumption specifications, *spec*, is given in Figure 5, where **action** is a program action, **action**$_{fun}$ is a value returning method call action, **name** is an introduced name, **type** is a program type name, and n and m are a pair of ordered natural numbers. Legal assumption specifications must also satisfy some additional constraints. Specifically, type expressions, $typeexpr$, may only involve **name**d variables introduced in an enclosing name scope that are type conformant with the **type**. Name initializations, $nameinit$, may only involve function call actions whose return type is conformant with the type of the initialized **name**. More generally, the use of introduced names in defining program actions must satisfy the type rules of Java.

$spec$	:==	**action**
	:==	$spec; spec$
	:==	$spec \mid spec$
	:==	$(spec)$
	:==	$spec?$
	:==	$spec*$
	:==	$spec+$
	:==	$spec + \{n\}$
	:==	$spec + \{n, m\}$
	:==	$<$ **type** $nameinit >: spec$
	:==	$< typeexpr$ **name** $>: spec$
$nameinit$	:==	**name**
	:==	**name** $=$ **action**$_{fun}$
$typeexpr$	:==	**type**
	:==	$typeexpr -$ **name**

Figure 5. Assumption Syntax

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 with the following specification:

```
environment {
 regular assumptions{
   <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`.

$r|s$ → `switch (chooseInt(2)) {`
`  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); }`

$< type\ name >: r$ → `{ type name = chooseClass(type);` $code$`(r);}`

$< type\ name = action_{fun} >: r$ → `{ type name = action`$_{fun}$`(); ` $code$`(r);}`

$< type - name_1 - \ldots - name_k\ name >: r$ → `{ type name;`
`  while (true) {`
`    name = chooseClass(t);`
`    if (name == name`$_1$`) continue;`
`    ...`
`    if (name == name`$_k$`) continue;`
`    break;`
`  }`
`  `$code$`(r);}`
`}`

Figure 6. Assumption Semantics

3.4 From Assumption Specifications to Code

Environment code generation can be separated into: action code generation and regular expression code generation. 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.

3.4.1 Action Code Generation

Action specifications usually involve approximation, for example, in the description of an assigned value or method parameter. Approximate values are specified using TOP values in action specifications. Code generation for TOP is dependent on type information, either the field type on the left-hand side of an assignment or the formal parameter type of a method call. For reference types C, TOP is translated as `chooseClass("C")`. For scalars, the translation in the case of booleans is `chooseBool()` and for other types the TOP token is preserved for use by subsequent abstraction tools [10].

For method calls, an additional complication in code generation is the need to resolve elided parameters, including the receiver object, and method name overloading. As explained in Section 3.2, the elided parameters are interpreted as TOP values and code is generated as described above. Treatment of name overloading is greatly simplified by query operations on the program's internal representation. One can specify a partial list of formal parameter types and the query returns a list of methods that match that signature. For example, a query for method `m` with an empty formal parameter type list on class C described in Section 3.2 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();
}
```

3.4.2 Regular Expression Code Generation

Regular expression assumption specifications are mapped to Java using the templates shown in Figure 6; 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.

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 select an instance to be bound to the name at the point where the name binding action is embedded in the

regular expression. The last three forms in Figure 6 illustrate the Java fragments that achieve local name introduction for non-deterministic binding, binding based on a function return, and restriction of the universe over which non-deterministic binding is performed. Global name scopes are implemented as variable declarations that are global to the body of the driver `run()` method.

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.

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 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

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 [7] 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, $\texttt{Act}$ is an alphabet of actions and $R \subseteq S(V) \times \texttt{Act} \times S(V)$ is a transition relation. We write $s \stackrel{a}{\longmapsto} s'$ 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. The labels on transitions can represent variable assignments, variable tests, and actions modeling transfer of control to and from a procedure.

A system may interact with its environment through shared variables and actions. V is partitioned into V^{int}, the set of *internal variables* (that only the system itself may modify), and V^{com}, the set of *common (shared) variables*. $\texttt{Act}$ is partitioned into $\texttt{Act}^{int}$, that denotes the set of *internal actions* (a symbol representing an internal action of a system is in the alphabet of only that system), and $\texttt{Act}^{com}$, that denotes the set of *communication (or interface) actions*.

Let $P_1 = \langle S(V_1), \texttt{Act}_1, R_1\rangle$, $P_2 = \langle S(V_2), \texttt{Act}_2, R_2\rangle$. We say that P_1 and P_2 are *compatible* if both their sets of internal variables and sets of internal actions are disjoint.

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 interface actions and asynchronously interleave all other actions. The internal variables of P_i may be modified only by the actions of 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.

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. Environments can represent both stubs and drivers.

4.2 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 $P_1 = \langle S(V_1), \texttt{Act}_1, R_1\rangle$ and $P_2 = \langle S(V_2), \texttt{Act}_2, R_2\rangle$ be two systems. We say that P_2 is a *sound abstraction* of P_1, denoted $P_1 \preceq P_2$, iff there is a simulation from P_1 to P_2.

A *simulation* [18] from P_1 to P_2 is a pair (ρ_s, ρ_a) of relations with $\rho_s \subseteq S(V_1) \times S(V_2)$ and $\rho_a \subseteq \texttt{Act}_1 \times \texttt{Act}_2$ such that if $(s_1, s_2) \in \rho_s$ and $s_1 \stackrel{a_1}{\longmapsto} t_1$, then there exists some state $t_2 \in S(V_2)$ and some action $a_2 \in \texttt{Act}_2$ such that $s_2 \stackrel{a_2}{\longmapsto} t_2$, $(t_1, t_2) \in \rho_s$ and $(a_1, a_2) \in \rho_a$.

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 $P_1 \preceq P_2$, then, for every universal temporal formula ϕ, $P_2 \models \phi$ implies $P_1 \models \phi$. However, if $P_2 \models \phi$ does not hold, it does not

mean that $P_1 \models \phi$ is necessarily false (i.e., completeness is sacrificed).

BEG synthesizes environment models from user specified assumptions written as regular expressions or LTL formulae. Let A be an environment assumption, and let E_{abs} be the transition system corresponding to the environment generated by BEG. In [21], we explain the synthesis of an environment model from LTL assumptions and we show that this environment model is a sound abstraction of actual environments that conform with the assumption, i.e., for any environment E that satisfies A, $E \preceq E_{abs}$. A similar argument applies to assumptions given as regular expressions, where the transition system (E_{abs}) that corresponds to the environment code synthesized by BEG is the finite state automaton that accepts the same set of strings as the regular expression [1].

Alternatively, BEG uses actual environment implementations (E) to extract assumptions and to build from them environment models (E_{abs}), and in [24], we show that these models are sound abstractions of the actual environments, i.e., $E \preceq E_{abs}$.

Proposition 4.1 *Given system P, environment assumption A (or actual environment implementation E), and universal property ϕ that we wish to check for P, such that $E \preceq E_{abs}$ and $E_{abs}||P \models \phi$. Then $E||P \models \phi$.*

Proof. It follows from the fact that generated environments E_{abs} are sound abstractions of actual environments E and that simulation preserves universal temporal properties. □

In other words, if the result of model checking a universal temporal property for a program that uses environment models generated by BEG is true, then the property holds for the program that uses the actual environment implementation, since the environment models are sound abstractions of the real environments. Note that the user specified assumptions have to be discharged on the actual environment implementations (i.e., E should satisfy A in order for $E \preceq E_{abs}$ to hold).

5 Experience with BEG

BEG is implemented using the SOOT framework [27]. BEG uses SOOT's symbol table, control flow graph, and a 3-address SSA-like bytecode representation, Jimple, to perform its analyses (i.e., the analysis of the unit code to discover the unit interface and the analysis of the environment implementations). The tools produce Java code as output that includes calls to the modeling methods introduced in Section 2. 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 JPF or Bandera.

5.1 Case Studies in Environment Generation

We have applied BEG to a variety of examples[1]. 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 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 coordinated 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 synthesis 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 realistic programming frameworks, we consider two non-trivial Java programs. **Autopilot** is a swing-based GUI for an MD-11 autopilot simulator used at NASA [25]; it is a framework client application. **ReplicatedWorkers** [12], is a parameterizable parallel job scheduling framework.

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

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

than 3600 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 system was checked for *mode confusion* by encoding a model of a user's understanding of the aircraft state. That user 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 user 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 assumptions about user behavior. We restrict our attention here to the generation of the drivers, for a more complete description see [25].

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 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 user 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 assumptions in terms of those mnemonics. Model checking the `Autopilot` class with the generated environment using JPF produced the following counter-example:

```
init; 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 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.

```
environment{
  instantiations{
    MyEvent incrMCPAltEvent = new MyEvent(400, 110);
    MyEvent incrMCPVSEvent = new MyEvent(540, 110);
    MyEvent pullAltKnobEvent = new MyEvent(420, 130);
    MyEvent flyEvent = new MyEvent(550, 440);
    1 User(new Autopilot());
  }
  definitions{
    pullAltKnob=mouseReleased(pullAltKnobEvent);
    incrMCPAlt=mouseReleased(incrMCPAltEvent);
    incrMCPVS=mouseReleased(incrMCPVSEvent);
    fly=mouseReleased(flyEvent);
    userExp=getUserExpectation();
  }
  regular assumptions{
    init(); incrMCPAlt*; pullAltKnob;
     (userExp; fly)*; incrMCPVS*; (userExp; fly)*
  }
}
```

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**

5.3 Replicated Workers

ReplicatedWorkers(RW) is a configurable framework designed to support the parallelization of simulations. In previous work [11], 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 use *inversion of control* to simplify application programming. This pattern means that program threads are created internally within the framework. Clients control the degree and asynchrony of parallelism in the framework 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 [11] using Bandera with SPIN [16] at the back end and were able to reproduce

the results from the study with one difference. We checked a framework instance under the environment defined in Figure 8 for deadlock and 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., [2]). 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.

The approach to environment generation from specifications presented in this paper builds on the work of Avrunin et al. [3] who developed tool support for analyzing partially implemented real-time systems whose components were implemented in Ada or specified using graphical interval logic and regular expressions. In addition to treating Java programs, we have customized the specification languages to better suit the goal of environment modeling and we extract environment models from existing code.

Another modular approach to checking multi-threaded programs is implemented in Calvin [13]. Their approach is aimed at procedure checking and relies on user specifications of environment assumptions that describe other procedures in the system and constrain interactions among threads. 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.

Recent work on generation of environment assumptions for *optimistic* environments has been described in [5, 14]. 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 can play an important role when using model checking as a kind of *unit testing* approach on existing code bases.

There are a number of examples of applying static analysis techniques in support of modular analysis or verification. Verisoft incorporates a static analysis to the closing of open systems by calculating the influence of externally defined data [6]. 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 [23] 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 leaves the values weakly specified, via TOP, and relies on subsequent abstraction phases to partition those values prior to model checking.

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://bandera.projects.cis.ksu.edu`.

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, by NSF under grant CCR-0306607, and by Intel Corporation under grant 11462.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[2] 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 Interna-*

tional Conference on Software Engineering, pages pp. 835–836. IEEE Computer Society Press, 2001.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[11] M. B. Dwyer 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.

[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] 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.

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

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

[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 real-time scheduling 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 1680)*, Sept. 1999.

[23] 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.

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

[25] 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*, Oct. 2002.

[26] 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.

[27] 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.

[28] W. Visser, G. Brat, K. Havelund, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, Sept. 2000.