Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science 144 (2006) 107–132
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Checking Event-Based Specifications in Java Systems

Steven P. Reiss[1,2]

Department of Computer Science
Brown University
Providence, RI 02912 USA

Abstract

One of today's challenges is producing reliable software in the face of an increasing number of interacting components. Our system CHET lets developers define specifications describing how a component should be used and checks these specifications in real Java systems. CHET is able to check a wide range of complex conditions in large software systems without programmer intervention. It does this by doing a complete and detailed flow analysis of the software and using this analysis to build a simpler, model program. This paper explores the motivations for CHET, the specification techniques that are used, and the methodology used in statically checking that the specifications are obeyed in a system.

Keywords: Specifications, model checking, flow analysis, finite-state properties, component usage.

1 Introduction

Much of software engineering is concentrated on ensuring the reliability of software. Work in this area includes safer languages, contracts, and tools for finding specific problems such as buffer overflow. Most of this work is limited in that it considers only the local execution behavior of a system. Software model checking takes the global behavior into account, but has typically only been used to prove relatively simple and specific properties of software. While

[1] This work was done with support from the National Science Foundation through grants CCF0218973, ACI9982266, CCR9988141. Shriram Krishnamurthi and Kathi Fisler provided significant advise and feedback.
[2] Email: spr@cs.brown.edu

doi:10.1016/j.entcs.2006.01.008

these efforts are helpful, they fail to address many of the problems of modern software development.

1.1 Software Properties

The goal of our work was to develop a tool that would check that components are used correctly in large software systems. We wanted a tool that would take a specification of how a component should be used along with the system and would then identify locations where the component was used incorrectly without any programmer intervention. While designing and implementing such a tool we realized that the same mechanisms could be used to check a wide variety of safety properties such as design patterns, user classes, and proper use of the programming language itself.

Examples of the properties we are able to specify and the tool is currently able to check include:

- For each use of an iterator in a program, *hasNext()* is called exactly once before *next()* is called.
- Iterators are not used while the structure is being modified outside of the iterator (concurrent modification).
- Callbacks are registered before a widget is shown.
- Each file opened by the application is closed.
- An XML writer component is used such that begin and end pairs match up and fields are added before any subelements.
- For a Java byte code library component, a current function is registered before any attempts to find a line number for an instruction are made.
- For a web crawler, each page that is found is processed correctly by either indicating error, providing a redirected URL, or by providing the page contents and the internal text to a URL support library.
- User defined errors (subclasses of *java.lang.Error*) thrown by the application are caught.
- Each class that represents an instance of a singleton design pattern [18] has only one associated object.
- That a chain-of-responsibility design pattern [18] is followed correctly.
- An object of class A is always locked before an object of class B.
- That the matrix stack in JoGL is used correctly.
- That the calls for setting up polygons in JoGL are used in the proper sequence.

- A session id is successfully obtained from a web service before the action methods are called.

1.2 Tool Requirements

In order for a tool that addresses properties of this sort to be successful it must meet certain requirements.

First, the properties must be easy to specify in a natural way. We want to encourage the use of the tool both for checking components and for checking internal properties of programs. If creating specifications is overly complex, programmers won't bother to create them.

Second, the tool needs to check each instance of the property in the program. A complex system can use a lot of iterators, files, errors, etc. Yet the tool should check each such use separately. This is the only way that meaningful information can be returned for each property.

Third, the identification of instances of the property in the program has to be automatic. One cannot expect a programmer to annotate or identify the hundreds or thousands of instances that occur in a moderately sized system. Such additional work will discourage the programmer and result in the tool not being used.

Fourth, the tool has to be fast enough so that programmers would use it everyday. Such a tool is most useful when it can be applied during program development. Our goal here has been to produce something that was not significantly slower than compilation.

Finally, the tool had to be as accurate as possible. First it has to be sound in that it must correctly identify each instance of the property in the program that does not meet the specification. Second it has to minimize false positives, instances that are identified as potential failures where failure does not or cannot occur.

In addition to creating positive requirements, our emphasis on component checking also lets us make certain simplifying assumptions. Components are generally used via method calls and it is most often the sequence of method calls that is important in determining component usage. Data, in the form of variables or fields, is generally of secondary importance in determining the set of such sequences and in the specification of valid component usage. This lets us make certain simplifying assumptions, notably, that we can ignore most data fields during our analysis. This greatly reduces the state space that needs to be explored and helps to make our analysis practical.

1.3 Results

We have developed a system, CHET, that meets these requirements. The system has been used to check the above and other properties on a variety of software systems including simple test programs, small (1-3 Kloc) student programs from a software engineering course, our current programming system (of which CHET is a part; a total of about 68 Kloc), and a variety of open source systems of up to 100Kloc). In these tests the system has proven accurate and robust and has demonstrated that the techniques we use are scalable and practical for large systems.

Performance, while not matching compilation speed, is quite acceptable. For our own system (68,000 lines of source, 370,000 analyzed byte codes), CHET takes about 12 minutes to identify over 540 specification instances from within the project and check each of these instances individually. Most of the smaller systems are checked in under a minute.

1.4 Paper Outline

The key to our system is a specification model that is flexible, powerful, general enough to define the various properties, where instances can be quickly identified, and where the properties can be efficiently checked. We start with an example of a specification in Section 2. We define CHET's specification model in Section 3 and related these to the actual code in Section 4. Section 5 provides a partial formalization of this model. Section 6 then describes how our tool checks the specification using a combination of flow analysis and model checking. Section 7 provides a comparison to related work. Finally, Section 8 summarizes our experiences in using the tool by providing statistics about running CHET on a variety of systems.

2 An Example

One problem with a Java program is the potential for concurrent modification errors. These occur when an iterator is in use and the structure it is iterating over changes. We want our system to check that such errors cannot occur.

CHET uses automata driven by parameterized program events to express the various conditions. Figure 1 shows a simple automaton for checking for comodification in vectors. (note that self-arcs have been suppressed). The CHET user can define this directly using an XML file that defines the states, events, parameters, and transitions, or using a graphical editor. The *Start* state represents the beginning of the program. An allocation of a *Vector* causes us to enter the *Play* state. We can modify the vector in this state but

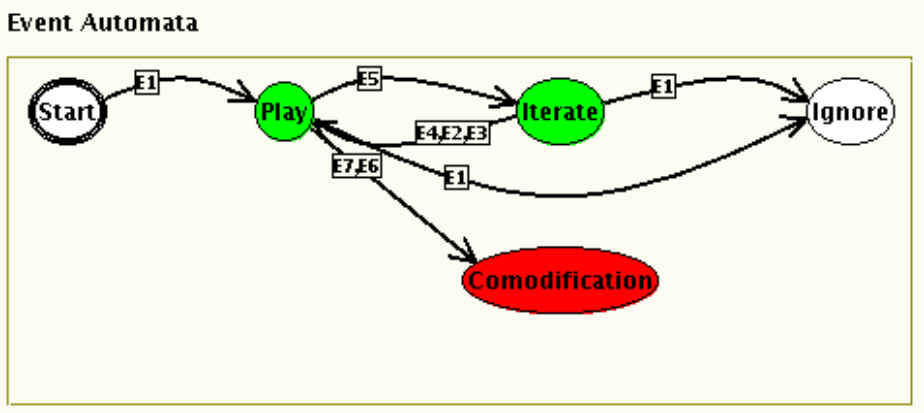

Fig. 1. Comodification specification.

if we try to iterate, we get an error. Calling the *Vector.iterator* method on the vector yields an *Iterator* object and puts us in the *Iterate* state. Here we can call *Iterator.next*, but any attempt to change the vector causes us to go back to the *Play* state. The *Ignore* state handles cases where the vector is reallocated. Note that this specification is parameterized so that it is meant to apply to each instance of a *Vector* created in the program and the call to *iterator* or the modification methods have to apply to that particular instance. Moreover, the object returned by the call to iterator is also a parameter and is used to determine which calls to *Iterator.next* are relevant.

The seven events associated with this specification and their parameters are shown in Figure 2. The first event occurs when a vector is allocated. The parameter represented by C1 for this event is defined as the result of the allocation. The next three events represent ways of modifying the *Vector*. The use of the parameter here ensures that these calls refer to the instance of the *Vector* that whose allocation we are considering. Additional events could be defined to check for other modifications. Event E5 occurs when a new iterator for this *Vector* is created. It is restricted by its dependence on C1 and defines a new parameter, C2, reflecting the *Iterator* object. The final two events then represent uses of the iterator and are restricted to calls where the object represented by C2 could be the *this* parameter. Note that CHET generates and checks an instance of the specification for each potential pair of Vector-Iterator (C1-C2) instances it finds in the code.

3 Specifications

The specification model in CHET uses extended finite state automata over parameterized events. Finite automata are relatively easy to define for the

Event	Type	Parameter
E1 (New)	ALLOC(Vector)	C1 = result
E2 (Add)	CALL(add)	this = C1
E3 (Add1)	CALL(addElement)	this = C1
E4 (Add2)	CALL(addAll)	this = C1
E5 (Iter)	RETURN(iterator)	this = C1 C2 = return
E6 (Next)	CALL (next)	this = C2
E7 (Next1)	CALL (nextElement)	this = C2

Fig. 2. Events in the specification of Figure 1.

various properties and are generally well understood by programmers. We use an extended form that uses bounded local variables to simplify nested specifications. The automata are driven by events representing program actions or states. The set of available events is central to both specifying and identifying properties. A graphical editor is available for the programmer to define such specifications.

3.1 Events

Events represent the basic actions of the program relevant to a particular property. In order to express a broad range of properties, we needed to have a variety of actions. For components, most of these actions relate to calling methods of the components. Other relevant actions include creating a component, setting fields, handling exceptions, and locking.

To support automatic identification of property occurrences, we need to have events describing what is going on at run time that can be detected statically from the code. More importantly, we must restrict these events to a *particular occurrence of a property*. This is done by having the events be parameterized. Parameters on events are used both to define and limit their occurrence. For example, we can look at the use of a particular *Iterator* in the program by associating a parameter with a particular allocation of the iterator and then only looking at the calls to the methods *next* and *hasNext* that can be have that particular object as the *this* parameter. Noting that the problem of identifying specific events in this context is similar to that faced by aspect-oriented languages we defined our initial event set based on those used in Aspect/J [27]

The set of events that we currently provide include:

- CALL events that can be parameterized by a combination of the *this* object, an argument of the method, or the *this* object of the calling method.
- RETURN events that can be parameterized by the *this* object of the call or the value being returned.
- FIELD events occurring when a given field is assigned a given value. The event can be parameterized by the object containing the field.
- ALLOC events triggered by an allocation and parameterized by the object being allocated.
- CATCH and THROW events parameterized by the appropriate object.
- LOCK and UNLOCK events parameterized by the object being synchronized.

This set is sufficient for defining a wide range of specifications. In addition, the system is designed to be extensible so that new event types can be added as long the instructions generating the event can be determined using flow analysis. Adding a new event type involves defining a new subclass for the event and adding the appropriate call to generate an instance of the class. The current events are implemented in 100-200 lines of code.

3.2 Using Event Parameters

Parameters are associated with events using a full interprocedural flow analysis of the program. This analysis is based on sources. A *source* is a representation of a value that has a specific creation point. For each source, our flow analysis computes all points in the program to which the source can flow. We use several types of sources. *Local sources* represent objects created directly by the code. Each new operator creates a new local source of the corresponding type. Arrays are represented as specialized local sources. *Fixed sources* are used to represent values that are created implicitly, either by the run time system or by native code. Finally, *model sources* are those that we add specifically to identify instances of a specification.

Each specification must include one event that is marked as a *trigger*. Each instance of this event in the system identifies an occurrence of the corresponding property. We introduce a new *model source* to uniquely identify each static instance of a trigger event. During flow analysis, this source associated with the value corresponding to the parameter of the trigger event. The flow of this source then is used to determine what other events are relevant to the property. For example, a RETURN trigger event identifies a new model source representing the value returned at each associated call. CHET defines

an occurrence of the specification for each such instance; the uses of the corresponding source would then identify the CALL or FIELD events that were relevant to the particular occurrence.

To allow multiple parameters in a specification as in the example of Figure 1, we associate a parameter mode with each parameter. The mode for a parameter that is not being created is CHECK. A parameter mode of NEW is used to provide an initial value for additional parameters. Other modes allow redefinition or reuse of a parameter.

Where multiple parameters occur, CHET computes all possible sets of occurrences. It first does a topological sort of all NEW parameters based on the specification. It uses the trigger event to define an initial partial occurrence, and then uses flow analysis to find all locations where the second parameter can be used, forming a new (potentially partial) occurrence for each. This procedure is followed for any additional parameters.

The flow analysis does a symbolic interprocedural execution of the whole program (including libraries). This analysis tracks the possible sets of sources on the stack, in local variables, and in fields and arrays at each point in the program. The sets of sources are represented as *values* which include information about the data type, whether the value can or must be null, the actual set of sources, and, for numerics, an optional range.

The analysis accurately tracks field and array values, selectively inlines methods based on the calling parameters, and models key Java data structures including hash tables and vectors. It also is able to handle the complexities of Java including native code, exceptions, threads, callbacks, dynamic loading and binding using reflection, and the implicit execution semantics of Java such as calls to static initializers and implicit field initializations.

The flow analysis is a conservative approximation. It ensures that if there is an execution where a source can flow to a particular point in the program, then the source will be associated with that point. It is conservative in that sources will be associated with points even in cases where no possible execution could result in that association.

3.3 Automata

Each property specification defines the set of relevant events, the set of event parameters, and an automaton over the events. The automaton consists of uninterpreted but labeled states and event-based transitions. Each automaton has a unique starting state. Each state can be tagged with a property indicating whether ending in this state represents an error, success, ambiguous, or don't care.

Transitions between the states consist of an event, a condition, and a set of actions. The condition and actions are based on automata variables which can range over bounded integral values. This extension to normal automata lets us specify limited instances of what would otherwise be context-free properties. For example, we have used them to specify that the number of entries and exits match for the XML writer class, assuming that we never nest the XML more than a finite amount.

4 Abstract Programs

To efficiently check each occurrence of a specification in the user's application, we generate an abstract program that models the application's behavior with respect to that particular occurrence. The abstract program is restricted to include only a simplification of that part of the code that can generate events relevant to the occurrence. These programs are generated in a separate phase that takes advantage of the information gleamed from the flow analysis.

This abstraction ensures that if there is an execution of the actual program exhibiting a certain sequence of specification events, then there is an execution of the abstract program generating the same sequence. This is conservative in that the abstract program may generate sequences that can never be exhibited in the actual program.

The abstract program consists of a set of routines. Each routine is composed of nodes and arcs similar to an automaton. There are actions associated with each node, but the arcs are uninterpreted. The associated actions control the behavior of the program and the generation of events. The current actions include:

- Enter a routine (**Enter**).
- Exit a routine (return or end of program) (**Exit**).
- Call a routine (**Call**).
- Generate an event for the particular property being checked (**Event**).
- Set a variable or return value to a given value (**Set**).
- Test a variable or return value (**Test**).
- Asynchronous call (e.g. start of a thread) (**ACall**).
- Begin synchronization for a set of sources (**Synch**).
- End synchronization for a set of sources (**Esynch**).
- Wait or timed wait on a set of sources (**Wait**).
- Notify or notify all on a set of sources (**Notify**).

- Do nothing (**Nop**).
- Start a callback thread (**CallOnce**)

The first step in building the abstract program is to determine the set of fields to be considered. This is done by starting with those explicitly mentioned in the specification and then using the flow analysis to determine heuristically what other fields are particularly relevant. Adding fields here helps remove false positives at the cost of increased analysis time, but the lack of fields doesn't affect the correctness of the approach.

The abstract program is built in four stages. First, it finds the location of all the events that define new parameters for the test. For simple tests with a single trigger event, this is trivial since it is based on the definition of the check. For more complex tests such as comodification, this requires a quick pass over the byte codes of the application.

Next CHET makes a pass over the byte codes to determine any fields that might be relevant to the check. Here it looks primarily for fields that can affect the control flow relevant to generating events.

Third, CHET makes a quick scan over the program to determine what routines are relevant to this particular abstract program and what routines can be ignored. This is done by first looking at each routine to determine if it can generate events relevant to the specification. These are the base routines that need to be included in the resultant program. Then it uses the call graph built during the flow analysis pass to determine what other routines need to be included. This check is fast and typically eliminates 50

Finally, the actual abstract program is generated by walking over the byte codes for each routine using a path-sensitive analysis that tracks the values of local variables. The system builds the abstraction for each routine, and then simplifies the overall abstract program by eliminating unnecessary routines (those that aren't used), calls to routines with no actions, and nodes that do nothing. The difficult problem here is identifying when the events associated with the specification may be generated.

Each type of event can be tied to specific instructions in the application code. CALL and RETURN events may be generated by invoke instructions for a particular method; ALLOC events may be generated by new instructions for a particular class; FIELD events may be generated by *putfield* or *putstatic* instructions for a particular field; THROW events may be generated by *athrow* instructions; ENTRY events may be generated at the first instruction of a routine; CATCH events may be generated at the first instruction of a catch block; LOCK events may be generated either on a *monitorenter* instruction or the first instruction of a synchronized method; UNLOCK events may be generated either on a *monitorexit* instruction or a return from a synchronized

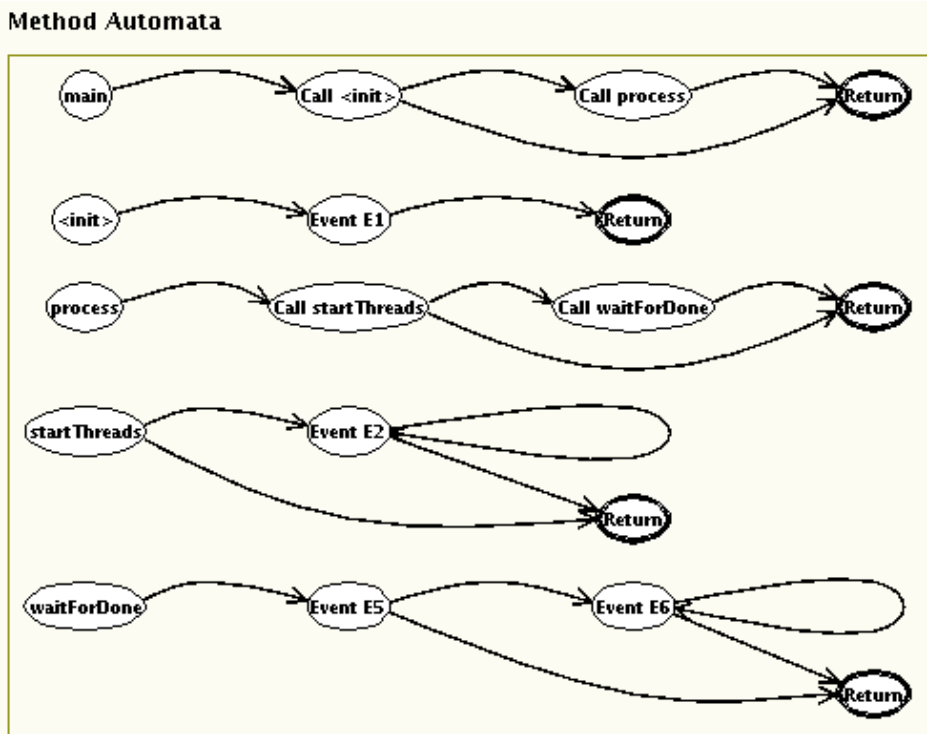

Fig. 3. The generated abstract program.

method. The real problem here is identifying the actual events using the event parameters.

This problem is addressed by maintaining a set of associated sources for each event parameter. The set is initially defined by the choice of the occurrence. The trigger event parameter is associated with the model source; any additional new parameters are associated with the set of sources that were identified by flow analysis at the location identified for the secondary event.

For each potential event occurrence, we look at the set of sources for that event and compare them to the set associated with the check. Normally, the mode of the parameter is CHECK. Here the possible sources for an instruction need to include the sources previously associated with the parameter. If they do, then the intersection of the two source sets becomes the parameter source set. Note that intersection is correct here in that we are generally attempting to isolate a single source from all the rules.

Figure 3 next shows the abstract program generated for one instance of this specification of Figure 1. The relevant source code from which this abstract program is generated is shown in Figure 4 and Figure 5.

```
public static void main(String [] args)
{
   CrawlMain cm = new CrawlMain(args);
   cm.process();
   System.exit(0);
}

private CrawlMain(String [] args)
{
   num_threads = CRAWL_NUM_THREADS;
   url_file = CRAWLER_URL_FILE;
   do_test = true;
   do_create = false;
   start_at = 0;
   url_count = CRAWL_DEFAULT_URL_COUNT;
   active_count = 0;
   page_total = CRAWL_DEFAULT_TOTAL;
   pages_queued = 0;

   work_sema = new Object();
   work_queue = new HashSet();
   urls_sema = new Object();
   urls_done = new HashSet();
   future_sema = new Object();
   future_urls = new HashSet();
   thread_set = new Vector();

   scanArgs(args);

   url_manager = CrawlerFactory.createUrlManager();
   url_manager.setDoTesting(do_test);
   if (do_create) {
      try {
         url_manager.clearRepository();
       }
      catch (IOException e) {
         System.err.println("Error with repository: " + e);
         System.exit(1);
       }
    }
}
```

Fig. 4. The methods from which the abstract program is generated (part 1).

```
private void process()
{
   active_count = 1;    // don't let things terminate

   System.err.println("CRAWL: Start processing");
   startThreads();
   loadUrls();

   synchronized(work_sema) {
      active_count -= 1;           // we are no longer active;
      if (active_count == 0) work_sema.notifyAll();
    }

   waitForDone();
   url_manager.addFutureUrls(url_file,future_urls);
   System.err.println("CRAWL: Done processing");
}

private void startThreads()
{
   for (int i = 0; i < num_threads; ++i) {
      CrawlThread ct = new CrawlThread(this,i);
      thread_set.add(ct);
      synchronized(work_sema) {
         ++active_count;
         ct.start();
       }
    }
}

private void waitForDone()
{
   for (Iterator it = thread_set.iterator(); it.hasNext(); ) {
      CrawlThread ct = (CrawlThread) it.next();
      try {
         ct.join();
       }
      catch (InterruptedException e) { }
    }
}
```

Fig. 5. The methods from which the abstract program is generated (part 2).

5 Formal Model

A better understanding of exactly how our system understands specifications and generates an abstract program for a particular specification can be obtained by formalizing the process. We start by considering specifications. A specification is an automaton with uninterpreted states, driven by a sequence of events. It can be viewed as a 5-tuple:

$$(\Sigma, S, E, P, M)$$

where Σ is the set of states, $S \in \Sigma$ is a distinguished state called the start state, E is the set of events, $P \subseteq \Sigma \times E \rightarrow \Sigma$ is the set of transitions, and $M \subseteq \Sigma \rightarrow A, X, I, Q$ is a mapping indicating for each state whether it is an accepting state, an error state, an ignore state, or unknown.

The interpretation of a specification for a sequence of events $E_1, \ldots, E_n$ is the result of applying P consecutively beginning with the start state to each E_i in turn. The result of this interpretation is found by applying M to the final state.

Understanding how the specification is related to the program is more complex. We start by considering the application code as a sequence of instructions, $C = (c_1, c_2, \ldots, c_n)$. Let R be a the set of routines in the program. We assume that each c_i is associated with a unique routine and that each routine $r \in R$ has a unique starting point c_r in the code. We define the non-call control flow of the application in terms of C as the relation F where $(c_i, c_j) \in F$ if and only if c_i can immediately precede c_j in some execution and both c_i and c_j and are in the same routine.

From the application code we generate an abstract program which can be represented as the 6-tuple:

$$(\Pi, T, R, \Gamma, A, \Theta)$$

where Π is the set of states, $T \subseteq \Pi \times \Pi$ is the set of transitions, R is the set of routines from the code, $\Gamma \subseteq R \rightarrow \Pi$ is a mapping that identifies the start state for each routine, A is a set of actions, and $\Theta \subseteq \Pi \rightarrow A$ is a mapping that associates an action with each state.

For illustrative purposes, we only consider a basic set of actions A. These are **Call**[r] where $r \in R$, **Exit**, **Nop**, and **Event**[e] where $e \in E$, the set of specification events. The formalism extends naturally to the other abstract program actions by including the appropriate information as part of the state.

To define the mapping from the original code to the abstract program, we consider pieces of the abstract program. We define an abstract fragment as

the 5-tuple:

$$H = (\pi, t, \pi_s, \pi_e, \Theta_H)$$

where $\pi \subseteq \Pi$ is a subset of the states of the overall program, $t \subseteq \pi \times \pi \subseteq T$ is a subset of the transitions restricted to the local states, π_s and π_e are elements of π representing a unique starting and ending state for this fragment, and Θ_H specifies the actions associated with the states. These abstract fragments are used to build the overall abstract program using the mapping $G \subseteq c \rightarrow H$ which takes each instruction c_i and maps it into an abstract fragment H_i.

The mapping defined by G must capture the essence of the program. In particular it must insure that

(i) If c_i can invoke a routine r in any execution, then H must reflect this with an action **Call**[r].

(ii) If c_i can be associated with an event e in any execution, then H must reflect this with an action **Event**[e].

(iii) If c_i is the first instruction of a routine, the H must reflect this with an **Enter** action.

(iv) If c_i can result in a return from a routine r in any execution, then H must reflect this with an action **Exit**.

We need to have G generate a fragment since a particular instruction might invoke one of a set of routines (in which case the fragment is the OR of the calls), might generate multiple events, or might generate an event and then do a call (and then generate another event). The information needed to compute G comes both from the semantics of the instructions (i.e. the definition of the Java virtual machine), and from the results of the flow analysis which tells us which routines can be invoked at a particular instruction (using type analysis) and which events can be associated with an instruction (using sources). Using G, the initial abstract program is defined by:

(i) Defining Π as the union of all π, T_0 as the union of all t, and Θ as the union of the Θ_H.

(ii) Defining the mapping Γ by mapping each routine to the start state of the first instruction of that routine.

(iii) Defining T by augmenting T_0 with connections from the end state of each fragment H_i to the start state of each fragment H_j where $(c_i, c_j) \in F$.

Executions of the abstract program are then defined using this set of actions and considering all possible transitions. In particular, we define a possible execution of an abstract program as a sequence of tuples (π_i, k_i, o_i) where π_i represents the current state, k_i is a list of state representing the current stack, and o_i is the sequence of events representing program output to this

point. The tuples must obey the following rules (where <> is the empty list and $(a|B)$ represents the list obtained by putting the element a in front of the list B):

(i) $\pi_1 = \Gamma(r_0), k_1 =<>, o_1 =<>$ where r_0 is a start method of the program (e.g. *main*).

(ii) If $\Theta(\pi_i) =$ **Call**[r] then we simulate a call by setting $\pi_{i+1} = \Gamma(r), k_{i+1} = (\pi_i|k_i), o_{i+1} = o_i$.

(iii) If $\Theta(\pi_i) =$ **Exit** and $k_i =<>$ then the program execution is over.

(iv) If $\Theta(\pi_i) =$ **Exit** and $k_i =< \pi_\alpha|k >$ then we simulate a return with $(\pi_\alpha, \pi_{i+1}) \in T, k_{i+1} = k, o_{i+1} = o_i$.

(v) If $\Theta(\pi_i) =$ **Nop** then the next step must satisfy $(\pi_i, \pi_{i+1}) \in T, k_{i+1} = k_i, o_{i+1} = o_i$.

(vi) If $\Theta(\pi_i) =$ **Event**[e] then the next step must satisfy $(\pi_i, \pi_{i+1}) \in T, k_{i+1} = k_i, o_{i+1} = (o_i|e)$.

The actual abstract program is defined as a simplification of this program. The simplification involves removing all nodes that have an associated **Nop** action and are not starting nodes for a routine and removing all nodes with an associated Call Action that calls a routine which has only a Return node (or a Nop and a Return). It should be clear from the above definitions that if there is an execution of the original abstract program that generates an event sequence , then there is also an execution of the simplified program that generates the same event sequence.

The checking done by CHET is then predicated on the relationship between the original program and the final abstract program. Here we claim (proof depends on a detailed model of the JVM and is beyond the scope of this paper):

Claim 5.1 *If there exists an execution of the original program such that the events $e_1 \ldots e_n$ would occur in that order, then there exists an execution of the abstract program $(\pi_1, k_1, o_1) \ldots (\pi_\alpha, k_\alpha, o_\alpha)$ where $o_\alpha = e_1 \ldots e_n$.*

This implies that the checking is a conservative approximation in that if the program can possibly generate a sequence of events, then the abstract program is guaranteed to reflect that. In particular it implies that:

Claim 5.2 *If for all output sequences o_α of the abstract program, the final state of the specification automaton under that sequence is not an error state, then the specification is guaranteed to hold in all executions of the original program.*

6 Checking Specifications

The remainder of the CHET system involves doing this last check. That is, the system looks at all possible executions of the abstract program and effectively determines the final specification state resulting from the sequence of events generated by each execution. Given that the number of possible executions is very large, this is done using model-checking techniques and a dynamic programming approach where we effectively execute the abstract program and the checking automaton in parallel.

Many of the specifications we want to check only involve a single thread of execution. For example, it is rare for a program to create an iterator in one thread and use it in another. Realizing this, we treat the single threaded case differently from the multithreaded case.

For the single threaded case we do a form of context-free model checking [26,31].We determine the set of checking states that are reachable at each node of the abstract program. A checking state here consists of a state from the specification (e.g. *Play* from Figure 1) along with values for each of the monitored variables and the latest return value. Value settings are currently limited to {*Null*, *NonNull*, *Unknown*} for objects and either a specific value or *Unknown* for numerics.

Checking is done by determining for each routine and each possible checking state on entry to that routine, the set of checking states that are possible on exit using a worklist algorithm, and using these summaries whenever possible. To handle routines that may not return, we distinguish specification states for which all transitions go to the state itself. These states represent either error conditions or a desired target state. Whenever the simulation gets into one of these states, we simulate an immediate return from the current method. This ensures that if the program can reach one of these "final" states, the algorithm will detect it.

For the example of Figure 3, the algorithm determines that there are two possible final states assuming you start in the *Start* state executing *main*. In particular, you can end in state *Play* by invoking *¡init¿* and skipping *process*, or you can end in state Iterate by invoking ¡init¿ and process, *startThreads*, and possibly *waitForDone*.

To handle multiple threads, we create an abstraction for each thread as in [4,17]. We find all instances of threads by looking for asynchronous call nodes in the abstract program. We convert each instance into its own automaton based using an inlining process. We extend the single-threaded checking approach by adding the state of these automata to the checking state and handling the possible transitions within the automata when we determine the

next state.

Synchronization checking is optional here since it is essentially unsafe. However, in most of the programs we have looked at, once they are restricted to a particular specification, the approximate synchronization represented by the set of sources has accurately reflected the actual synchronization done by the program and hence yields a more meaningful abstract program.

6.1 Reporting the Result

The output from the above procedure is simply the set of possible ending states that can be achieved for a given main program. It does not include control flow information that indicates how a particular state can actually be attained in a run. Thus, it does not provide enough information for a programmer to understand why or how the program gets into these states. To provide this information, we augmented the framework to produce a trace of the execution to the point where the target state is reached.

This is done as a separate pass over the abstract program. This pass uses much of the same techniques as the checking pass, but does a breadth-first search over the executions while tracking calls and attempting to find the specified target state.

CHET generates output for each possible final specification state that is not specifically ignored. This output consists of a trace of the relevant branch points and calls from the original program that would result in the particular final state. A front end lets the user step through the resultant trace and see the corresponding source code. An example of this is shown in Figure 6.

7 Related Work

Checking properties of software systems has a long history that includes original attempts at proving software correct, extended compiler checking such as Lint [32], static condition checking as in CCEL [12], and verification-based static checking such as in LCLint [16]. More recently and more related to our work, there has been a significant body of work on software model checking [24].

Software model checking typically starts with a software system and a property to check. The software system is then abstracted into a representation that is more amenable to model checking by abstracting the original program into a much smaller program and then converting that program into a finite state representation. The various systems that have been developed differ in what they consider the software system to be checked, in the way they define the property to be checked, in the way they do abstraction, in how they

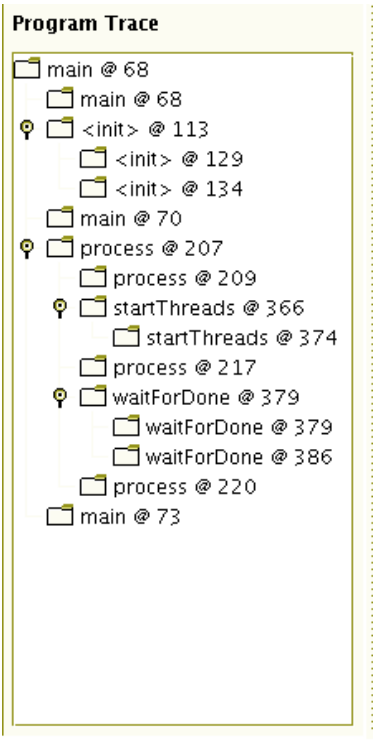

Fig. 6. Sample output trace.

map the program into a finite state representation, and in how they actually do the checking. Our tool combines aspects of a number of other systems to meet the requirements outlined above.

Most of the software model checking systems start with source code. For example, the original Java Pathfinder [19,20] translated Java programs into Promela, the input language for the SPIN model checker [23,24], while MAGIC starts by analyzing C programs to build a control flow graph [7]. Java Pathfinder 2 [5,30,36] instead starts from Java byte codes, essentially a binary representation, while the Bandera system [9,10,13] uses both the source and the byte code representation. The advantages of using the binary representation are that it is often simpler than the source, one does not have to worry about the vagaries of the language, and, most importantly, one can check not only the user's system but also the interactions between the user's system and libraries, a feature that is needed to analyze real programs. Thus we also start with byte code, using IBM's JikesBT package [28].

Finite state automata are the principal representations used for specifying properties. These are defined either directly or using a language that can be mapped into a finite state representation. The automata are triggered by

program events and it is the characterization of these program events that differentiates the systems. Most of the systems including Bandera and Flavers [8] require that the user explicitly define events or predicates in terms of the code for each item being checked, although the Bandera Specification Language allows parameterized specifications similar to what we can do. Other systems such as Metal [15] and ESP [11] use simple parameterized source code patterns which let the programmers specify all events of a given type with a single specification. SLIC [1] achieves the same effect using an event-oriented language. The MaC framework [29] takes a similar approach for specifying dynamic instrumentation using an event definition language. Patterns have also been used to simplify the definition of commonly occurring idioms in the specifications [14]. MAGIC uses labeled transition systems (LTS) to model procedures where the labels correspond to program statements. The Bogor framework uses the Java Modeling Language (JML). This consists of comments that contain pre and post conditions [33,34]. Our approach provides the automatic functionality of Metal or ESP using an event-based specification similar to MaC or SLIC. This lets us check the type of complex conditions that the latter tools can handle while providing the ease of use of the former ones.

One key to successful software model checking is the generation of small abstractions that reflect the property being checked without irrelevant details. The different approaches do this in different ways. The C2BP package within SLAM [2,3] and Java Pathfinder [35] convert the user's code into a Boolean program using predicate abstraction where each predicate relevant to the specification being checked is replaced with a Boolean variable. Bandera uses data abstraction to map the program types into abstract predicates that can be finitely modeled. Trailblazer looks only at control flow events and actions and eliminates all data [25]. ESP does a combination of control and data flow analysis to build a simplified version of the original program. Flavers constructs a trace flow graph by inlining control flow graphs of the various methods and adding arcs to represent synchronization events. Java Pathfinder 2 uses static analysis to reduce the state space by finding concurrent transitions [6]. Bandera, ESP, Java Pathfinder, and Flavers all use some type of slicing technology to restrict the abstraction to those portions of the program that are relevant to the conditions being checked. BLAST takes an additional step, using the verification process to identify what needs to be refined in the abstraction and building a new model based on this information [21]. Later work on BLAST uses Craig interpolation and proof techniques to better the abstraction [22]. MAGIC builds finite data abstractions based on the predicates being checked and uses these to augment a control flow

graph. Our approach to date is probably closest to that of ESP in that we use both control and data flow analysis. However, we limit ourselves to a small, heuristically chosen subset of the relevant variables, which greatly simplifies the abstraction in exchange for a loss of accuracy, and we achieve the effect of path-sensitive analysis using automata simplification techniques.

The various systems also differ in their representation of an abstract program for checking. Some of the systems actually generate an automaton. For example, Flavers inlines routines and adds synchronization arcs to build a single large automaton that can be checked, while MAGIC uses its program analysis to build a model representing the implementation that can be compared to the model representing the specification using model checking. Bandera and the first Java Pathfinder map the program into Promela, the input language for the SPIN model checker. Our approach is different. We generate an abstract program with calls, synchronized blocks, and events. This lets us handle complex and recursive programs easily and compactly. In addition, we use a Flavers-like automaton (still with calls, synchronized blocks and events) to represent the behavior of each program thread other than the primary one.

Checking in Bandera is done using external model checkers such as the SPIN model checker. SLAM and Java Pathfinder 2 use their own model checkers, SLAM's is based on Boolean programs, and Pathfinder's is based on a modified JVM [5,30]. Our approach has been to develop our own checker to match our program-like abstraction representation. The checker is unique in the way it handles routines and synchronization, and extends from a detailed single-threaded analysis to an approximate multithreaded analysis quite naturally.

The work described here is probably closest to that of ESP. Both use finite-state specifications to describe the problem, although ESP's are code-based while ours are event-based. Both use data flow analysis to limit the study of the program to the particular case at hand. Here the two differ in their approach. ESP does property simulation, using the specification information to drive the data flow. Given a set of variables from the specification, it determines which branches have any effect on those variables as part of flow analysis, merging paths that don't affect these variables or make other state transitions. Our approach is much simpler but potentially less accurate. First we do a generic data flow pass which considers, in a limited way, all variables and their values. Second, we do a cursory second data flow analysis to choose the variables that should be considered, with the only criteria that the variables are used directly in a conditional that determines whether an event is generated or not. Finally, we build a complete abstract program and then eliminate any irrelevant paths using an automata simplification approach. For

System	LOC	#BC	Proj #BC	flow time	Total Time
Onsets	2669	155645	6248	44.97	1:05.91
Crawler	3556	205514	5455	68.14	1:25.11
Pinball	11264	241264	54892	72.89	1:38.63
Freecs	20570	228163	50567	109.77	3:05.62
Taiga	51391	194923	15404	52.63	1:33.96
Egothor	54317	559898	268663	371.89	9:46.06
Clime	64998	421709	117575	255.85	13:06.20
Jalopy	94636	639310	348843	1697.79	31:32.11
Openjms	95470	308198	30787	384.53	7:19.74

Fig. 7. The results of running CHET on various systems.

most cases, the effect is the same – we only have to check paths in the program that are relevant to the specification.

8 Experience

We have written and text the specifications for the properties listed in the introduction in our specification language. The specifications tend to be small, all handled with fewer than 10 states and fewer than 12 different events.

The tables in Figure 7 and Figure 8 summarize some of our experience with CHET. The first column in both figures indicates the system: Onsets is a mathematical game based on sets, Crawler is a web crawler, Pinball is a 3D pinball program, Freecs is a shareware chat program, Taiga is a distributed programming system, Egothor is an open source text search engine, Clime is our constraint based programming environment which includes CHET, Jalopy is an open source Java pretty printer, and Openjms is an open source implementation of the Java message service.

The second through fourth columns of Figure 7 indicate the size of the systems, first in lines of code, then in number of byte codes analyzed overall and within the project respectively. Discrepancies here are generally due to uses of Java reflection that we did not detect in the open source systems. The fifth column gives the time in seconds for CHET's interprocedural flow analysis, while the sixth is the total time taken by CHET to analyze the system.

System	# Tests	# Errors	Avg Test	Max Test
Onsets	10	3	3.4	17.0
Crawler	25	11	3.5	14.0
Pinball	39	7	61.8	673.0
Freecs	68	2	10.7	452.0
Taiga	39	3	2.36	44.0
Egothor	152	27	5.7	66.0
Clime	571	20	14.0	2563.0
Jalopy	36	9	18.5	222.0
Openjms	54	45	2.24	30

Fig. 8. The results of running CHET on various systems.

The second and third columns of Figure 8 indicate the number of test instances that were identified and the number of errors that were detected. Relatively small numbers of tests for some of the systems here are generally due to the fact that while we correctly identify and do flow analysis for specification instances that occur during callbacks, we did not test such instances. The next two columns indicate the average and maximum time for a single test in milliseconds.

For several of these cases, most notably those involving Clime, we have gone through each individual test and checked manually whether the error reports were accurate. We have also manually checked in all of our systems that all instances of each of the particular tests were caught and analyzed by CHET.

The difference between the flow time and the total time is primarily the time it takes CHET to find the tests, generate the abstract programs, and then check each of the tests. Finding the tests involves identifying those routines that are not involved in any check, finding the relevant fields for each test using a simple flow analysis and finding instances of tests that involve multiple sources, again using a simple flow analysis. On most systems, this takes about half the time. Building the abstract programs in a path-sensitive fashion and checking these programs takes the rest. The program building pass is fast enough so that its time is dominated by the checking time.

The table shows that the flow analysis essentially dominates the performance of the system and the flow analysis is dependent mainly on the com-

plexity of the code and the number of byte codes in the project. Instances of the various specifications are found readily and accurately. Most instances are checked in around 10 milliseconds, with only a few outliers taking on the order of seconds. The bulk of the remaining time (about 20the set of fields to use during checking.

More notably, the table together with our manual checks show that the technology in CHET can find and check large numbers of instances of relatively complex software conditions in real Java systems both quickly and accurately.

CHET itself comprises about 30,000 lines of Java code. In addition, our visual front end and editor (from which the figures in this paper were taken), is another 7,000 lines. The code is available from

http://www.cs.brown.edu/people/spr/research/envclime.html.

9 Summary

The analysis has proven to be quite robust. Of the 540 checks that are done on our own system, it reports 33 errors, of which 15 are false positives. The rest are actual violations, mainly instances where exceptions that should never actually occur could cause the specifications to be violated.

Our work on CHET demonstrates that it is possible to effectively and automatically check significant properties in large software systems. Using specifications that consist of simple automata over parameterized events lets us check a variety of properties and provides, along with flow analysis, the information needed to both automatically isolate all instances of those properties and to generate easily-checkable abstract programs for each such instance. Our checking techniques provide a high degree of accuracy in the common single threaded case with relatively few false positives and extend naturally to the threaded cases.

References

[1] Ball, Thomas and Sriram K. Rajamani, *SLIC: a specification language for interface checking*, Microsoft Research Technical Report MSR-TR-2001-21, (2001).

[2] Ball, Thomas, Todd Millstein, Rupak Majumdar, and Sriram K. Rajamani, *Automatic predicate abstraction of C programs*, Proc. SIGPLAN 01, pp. 203-213 (June 2001).

[3] Ball, Thomas and Sriram K. Rajamani, *The SLAM project: debugging system software via static analysis*, Proc. POPL 2002, (2002).

[4] Bouajjani, A., J. Esparza, and T. Touilli, *A generic approach to the static analysis of concurrent programs with procedures*, POPL 2003, pp. 62-73 (2003).

[5] Brat, Guillaume, Klaus Havelund, Seung Joon Park, and Willem Visser, *Java PathFinder: Second generation of a Java model checker*, Proc. Post-CAV Workshop on Advances in Verification, (July 2000).

[6] Brat, Guillaume and WIllem Visser, *Combining static analysis and model checking for software analysis*, Proc. ASE 2001, pp. 262-271 (2001).

[7] Chaki, Sagur, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith, *Modular verification of software components in C*, IEEE Trans. on Software Engineering **30**(6) pp. 388-402 (June 2004).

[8] Cobleigh, J. M., L. A. Clarke, and L. J. Osterweil, *FLAVERS: A finite state verification technique for software systems*, IBM Systems Journal **41**(1) pp. 140-165 (2002).

[9] Corbett, James C., Matthew B. Dwyer, John Hatcliff, and Robby, *A language framework for expressing checkable properties of dynamic software*, SPIN 2000, pp. 205-223 (2000).

[10] Corbett, James C., Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng, *Bandera: extracting finite-state models from Java source code*, ICSE 2000, pp. 439-448 (May 2000).

[11] Das, Manuvir, Sorin Lerner, and Mark Seigle, *ESP: Path-sensitive program verification in polynomial time*, Proc. PLDI 2002, (June 2002).

[12] Duby, Carolyn K., Scott Meyers, and Steven P. Reiss, *CCEL: a metalanguage for C++*, Proc. Second Usenix C++ Conference, (August 1992).

[13] Dwyer, Matthew B. and John Hatcliff, *Slicing software for model construction*, Proc. 1999 ACM Workshop on Partial Evaluation and Program Manipulation, pp. 105-118 (1999).

[14] Dwyer, Matthew B., George S. Avrunin, and James C. Corbett, *Patterns in property specifications for finite-state verification*, Proc. ICSE 99, pp. 411-420 (1999).

[15] Engler, Dawson, Benjamin Chelf, Andy Chou, and Seth Hallem, *Checking system rules using system-specific, programmer-written compiler extensions*, Proc. 6th USENIX Conf. on Operating Systems Design and Implementation, (2000).

[16] Evans, David, John Guttag, James Horning, and Yang Meng Tan, *LCLint: a tool for using specifications to check code*, Software Engineering Notes **19**(5) pp. 87-96 (December 1994).

[17] Flannagan, C. and S. Qadeer, *Thread-modular model checking*, SPIN '03, pp. 213-225 (2003).

[18] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns", Addison-Wesley (1995).

[19] Havelund, Klaus and Jens Ulrik Skakkebaek, *Applying model checking in Java verification*, Proc. 5th and 6th SPIN Workshop, Lecture Notes in Computer Science **1680** pp. 216-231 Springer-Verlag, (1999).

[20] Havelund, Klaus and Thomas Pressburger, *Model checking Java programs using Java Pathfinder*, Intl Journal on Software Tools for Technology Transfer **2**(4)(April 2000).

[21] Henzinger, Thomas A., Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre, *Lazy abstraction*, Proc. POPL '02, pp. 58-70 (2002).

[22] Henzinger, Thomas A., Ranjit Jhala, Rpak Majumdar, and Kenneth L. McMillan, *Abstraction from proofs*, Proc. POPL '04, pp. 232-244 (2004).

[23] Holzmann, Gerard, "The Design and Validation of Computer Protocols", Prentice Hall (1991).

[24] Holzmann, Gerard J. and Margaret H. Smith, *Software model checking*, Forte, pp. 481-497 (1999).

[25] Holzmann, Gerard J. and Margaret H. Smith, *Software model checking: extractin verification models from source code*, Software Testing, Verification, and Reliability **11**(2) pp. 65-79 (2001).

[26] Hungar, Hardi and Bernhard Steffen, *Local model checking for context-free processes*, Nordic Journal of Computing **1**(3) pp. 364-385 (1994).

[27] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, *An Overview of AspectJ*, in European Conference on Object-Oriented Programming, (2001).

[28] Laffra, Chris, Doug Lorch, Dave Streeter, Frank Tip, and John Field, *What is Jikes Bytecode Toolkit*, http://www.alphaworks.ibm.com/tech/jikesbt, (March 2000).

[29] Lee, I., S. Kannan, M. Kim, O. Sololsky, and M. Viswanathan, *Runtime assurance based on formal specifications*, Intl. Conf. on Parallel and Distributed Processing Techniques and Applications, (June 1999).

[30] Lerda, Flavio and Willem Visser, *Addressing dynamic issues of program model checking*, Lecture Notes in Computer Science, Proc. 8th SPIN Workship **2057** pp. 80-102 (2001).

[31] Qadeer, Shaz, Sriram K. Rajamani, and Jakob Rehof, *Summarizing procedures in concurrent programs*, POPL 2004, pp. 245-255 (2004).

[32] Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan, *The C programming language*, Bell Systems Tech. J. **57**(6) pp. 1991-2020 (1978).

[33] Robby, Matthew B. Dwyer, and John Hatcliff, *Bogor: an extensible and highly-modular software model checking framework*, SIGSOFT FSE 2003, pp. 267-276 (2003).

[34] Rodriguez, Edwin, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby, *Extending JML for modular specification and verification of multi-threaded programs*, SAnToS Laboratory Tech Report TR2004-10, Kansas State University, (May 2005).

[35] Visser, Willem, Seung Joon Park, and John Penix, *Using predicate abstraction to reduce object-oriented programs for model checking*, Proc. ACM SIGSOFT Workshop on Formal Methods in Software Practice, (August 2000).

[36] Visser, Willem, Klaus Havelund, Guillaume Brat, and Seung Joon Park, *Model checking programs*, Automated Software Engineering Journal **10**(2)(April 2003).