CHET: A System for Checking Dynamic Specifications

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912
spr@cs.brown.edu

Abstract

Software specifications describe how code is suppose to behave. Software model checking and related activities statically investigate software behavior to ensure that it meets a particular specification. We have developed a tool, CHET, that uses model checking techniques to do large-scale checking of dynamic specifications in real systems. The tool uses a finite state specification of the properties to check in terms of abstract events. It first finds all instances in the system where this specification is applicable. For each such instance, it creates an abstract model of the software with respect to the events and then checks this model against the specification. Key aspects of CHET include a full interprocedural flow analysis to identify instances of the specifications and restrict the resultant models, and greatly simplified abstract programs that are easily checked. The system has been used to check a variety of specifications in moderate-sized Java programs.

1. Introduction

Many of the problems with software systems derive from the code not behaving as it is specified to do. In today's world of components and libraries, one of the most pressing problems is ensuring the libraries and components are used correctly. This is true both for system libraries and outside components as well as for the packages and internal components that comprise the system.

Our goal was to develop a framework where the behavior or usage of a class or package can be specified and then checked against a program. We wanted this framework to be practical and useful on real systems. It had to be able to take specifications from a variety of sources. It had to be fully automatic, taking a specification, finding all instances of that specification in the program, and then checking each instance. It had to be efficient so it can work with large programs. Moreover, to be useful, it had to be accurate, identifying all violations of the specifications with a minimum of false positives.

The framework we have developed meets most of these criteria. In the next section we discuss some of the related work in this area. The following sections describe our approach, first by providing the overall architecture and then going into detail on the various components. We conclude by discussing our experiences and future directions.

2. Related Work

Static checking of software systems has a long history that includes original attempts at proving software correct, extended compiler checking such as Lint [27], static condition checking as in CCEL [10], and verification-based static checking such as in LCLint [14]. More recently, there has been a significant body of work on software model checking [20].

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 map the program into a finite state representation, and in how they actually do the checking. Our system combines aspects of a number of other systems to produce a practical alternative that works on real programs.

Most of the software model checking systems start with source code. For example, the original Java Pathfinder [15,16] translated Java programs into Promela, the input language for the SPIN model checker [19,20]. Java Pathfinder 2 [4,25,29] instead starts from Java byte codes, essentially a binary representation, while the Bandera system [7,8,11] 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 easily check not only the user's system but also libraries and the interactions between the user's system and libraries. Thus we also start with byte code, using IBM's JikesBT package [23].

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 [6] require that the user explicit define events or predicates in terms of the code for each item being checked. Other systems such as Metal [13] and ESP [9] 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 [24] 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 [12]. Our approach provides the functionality of Metal or ESP using an event-based specification similar to MaC or SLIC.

The 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 the DEOS system [28] 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 [21]. 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 [5]. 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 a 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 [17]. Later work on BLAST uses Craig interpolation and proof techniques to better the abstraction [18]. Our approach to date is probably closest to that of ESP in that we use both control and data flow analysis. However, we do not attempt to find all 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 automata. For example, Flavers inlines routines and adds synchronization arcs to build a single large automata that can be checked. 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 automata (still with calls, synchronized blocks and events) to represent the behavior of each program thread other than the primary one. We do not attempt to generate a program that includes all the possibly relevant state variables on the assumption that this is too much in a large system. Instead, we include a minimal set of variables and let the specification indicate additional ones as needed.

Checking in Bandera is done using 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 [4,25]. 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.

3. Architecture

Our framework uses multiple passes to first find all occurrences of the given specifications in the user program, then to check each of those occurrences. The checking itself is done in two phases: the first builds a model of the program with respect to the particular instance of the specification and the second checks whether that model meets the specification.

We start with the code base of the program and a set of specifications. The code base is given as the set of Java class files that comprise the system, and involves not only the user's code but also all system and other libraries that are used by the program. The specifications are defined as extended finite state machines over program events. These are described in detail in the next section.

The first phase of the system does a full interprocedural flow analysis to find all the instances of each specified check and to identify the potential set of events for each instance. This is done by tracking sources in the code where a source is the trigger or variable used in a specification event. This is described in Section 5.

The result of the data flow is used to create a set of specification instances each of which is the combination of a single specification automaton and a single source of the appropriate type. For each of these instances, the system creates a abstract program consisting of a set of finite state machines each representing a particular method. The nodes of the machine represent actions that either generate the

events of the specification or model program behavior. The use of a set of machines here lets the system readily model recursion and complex systems. This process is described in Section 6.

Once a model is created, it is checked by effectively simulating all possible runs of the model with respect to the specification automata. This is done by determining for each method and each specification state that might apply at the start of executing the method, the possible specification states when the method returns. This is described in Section 7. The result of this provides the set of potential result states for the specification. A report with these results is then augmented with a trace of a program execution that exhibits the corresponding behavior as described in Section 8.

4. Specifications

We wanted a fairly generic specification mechanism that was able to model UML action diagrams, class and library usage rules, and the behavior part of design patterns. Based on an analysis of these different inputs, we settled on finite state machines utilizing parameterized events.

Because we didn't want to modify the application, the candidate events had to relate directly to what occurred in the program at run time. Moreover, we needed to have events that we could determine from a static analysis of the Java class files. This is a similar problem to that faced by aspect-oriented languages such as Aspect/J [22].

The set of events that we currently provide include:

- CALL events that are triggered by a call to a particular method or optionally any method that redefines it. This event can be parameterized by a combination of the *this* object, the first parameter to the method, or the *this* object of the calling method.
- RETURN events that are triggered by the normal return from a particular method. This can be parameterized by the *this* object of the call or the value being returned.
- ENTRY events that occur when a method is entered. These can be parameterized by the *this* object or the first parameter.
- FIELD events that occur when a given field is assigned a given value. The event can be parameterized by the object containing the field. If the value is numeric, the check can be equal or not-equal to an integer. If the value is an object, the value can be null or non-null.
- ALLOC events that are triggered by an allocation of an object of a particular class or optionally any of its subclasses.

These are 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 needed as long the instructions generating the event can be determined using flow analysis. For example, adding support for events representing throws, catches, or uses of an object could be done in under a day.

```
<CHECK META="generic" NAME="iterator_check" >
    <CLASSES>
        <USE ID="C1" CLASS="java.util.Iterator"/>
        <USE ID="C2" CLASS="java.util.Collection" />
    </CLASSES>
    <EVENTS>
        <EVENT ID="E1" NAME="NewIterator"
            TYPE="RETURN"
            VALUE="C1" METHOD="iterator"
            TRIGGER="1" CLASS="C2" />
        <EVENT ID="E2" NAME="hasNext"
            TYPE="CALL" OBJECT="C1"
            METHOD="hasNext" />
        < EVENT ID="E3" NAME="next"
            TYPE="CALL" OBJECT="C1"
            METHOD="next" />
    </EVENTS>
    <AUTOMATON START="S1" TRIGGER="NEW">
    <STATE ID="S1" ACCEPT="1" NAME="Start" >
        <ON EVENT="E1" GOTO="S2" />
    </STATE>
    <STATE ID="S2" ACCEPT="1" NAME="IteratorAllocated" >
        <ON EVENT="E2" GOTO="S3" />
        <ON EVENT="E3" GOTO="S4" />
    </STATE>
    <STATE ID="S3" ACCEPT="1" NAME="hasNextCalled" >
        <ON EVENT="E3" GOTO="S2" />
    </STATE>
    <STATE ID="S4" ERROR="1" NAME="nextCalled" />
    </AUTOMATON>
</CHECK>
```

FIGURE 1. Specification of Iterator behavior.

Automata are defined over these events using an XML notation that specifies the set of relevant events, defines the states, identifies the starting state, and then defines for each state and event, the appropriate transition. At least one of the events must be designated as a *trigger* event. This event is used to define an instance of the specification in the source using its corresponding parameter.

An example specification is shown in Figure 1. This specification starts by defining the classes to be used where each class usage represents a potential parameter. Then it defines three events. The first occurs whenever an implementation of the method Collection.iterator returns. It is a trigger and is parameterized by the returned value C1. This causes the return value to be associated with C1. The other two events occur when the methods *hasNext* and *next* are called on the value associated with C1. The automaton itself is fairly simple, essentially saying that *hasNext* must be called before *next* can be.

5. Finding Instances

The specification of Figure 1 says that for every iterator in the program the given condition must hold. The first phase of our framework finds all occurrences of the specification in the program so that each can be checked separately. We define an instance statically, creating one for each program location where the triggering event might

```
1    package spr.simple;
2    import java.util.*;
3
4    public class Simple {
5    public static void main(String [] args) {
6          Simple s = new Simple();
7          s.setupList();
8          s.printList();
9    }
10
11   private List    our_list;
12
13   public Simple()        { our_list = new Vector(); }
14
15   private  void setupList() {
16         our_list.add("Hello");
17         our_list.add("World");
18         our_list.add("Goodbye");
19   }
20
21   private void printList() {
22         for (Iterator it = our_list.iterator();
23               t.hasNext(); ) {
24               System.out.println((String) it.next());
25         }
26         int sz = our_list.size();
27         Iterator it = our_list.iterator();
28         for (int i = 0; i < sz; ++i) {
29               System.out.println((String) it.next());
30         }
31   }
32   }     // end of class Simple
```

FIGURE 2. Java program using Iterators

occur. For example, the program in Figure 2 has two instances of the iterator check, one for line 22 and one for line 27. CHET provides the option of looking for instances in the actual system code or in the system code as well as any libraries used. We generally restrict ourselves to looking at instances in the user's code.

We find specification instances by doing a full interprocedural data flow analysis of the program. The flow analysis has two objectives. First, it must identify and track the potential objects in the program that correspond to possible parameters from the specifications. This is needed to identify instances of the specifications and to locate the possible sources for the specification events. Second, it should simplify the next stages of static checking through the use of flow information. For example, it should let us correlate synchronized regions, identify the potential uses and values of relevant fields, determine which virtual methods are callable from each particular call site, differentiate static methods instances based on parameters, and determine code that can never be executed.

Flow analysis is based on sources. A *source* is a representation of a value that has a specific creation point in the program. For each source, our flow analysis computes all points in the program to which the source can flow. CHET uses several types of sources. *Model sources* are those that are developed from the specification events. For example, an ALLOC event identifies a new model source for any corresponding new statement; CALL and RETURN events can generate model sources for the *this* parameter, the first argument, or the return value; FIELD events generate a model source for each field access and set. *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. Finally, *fixed sources* are used to represent values that are created implicitly, either by the run time system or by native code.

The flow analysis does a symbolic execution of the whole program (including libraries) that 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 of values. While much of this intermediate information is discarded, information relating model sources to their corresponding locations is retained for later use. The algorithm uses a worklist to look at each method separately and requeue methods whenever an associated value such as the return from a called method or the contents of an accessed field changes.

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

Several flow analysis techniques are included to make the analysis both accurate and efficient. Field and array values are generally associated with their corresponding local sources. This means that the system can accurately track the set of values that a particular instance of a field can have. Because this is costly in general, the system also supports global field values and chooses between using a global and the local representation of the field based on parameter settings, for example, using local values for project fields and global values for library fields.

The analysis is also able to distinguish different instantiations of a method based on the values associated with the parameters, effectively doing inlining of methods at their different call sites. Using a different instance of a method for a call site ensures that one more accurately tracks the flow of values both from and to the parameters of that method. It also allows us to distinguish methods where the behavior is dependent on parameter flags. Again, this can be costly in general since it results in more code that needs to be analyzed and more states and values to track. The system chooses whether to create a new instance for a call site based on whether the caller or callee is a project or library method and whether any of the parameters contain a model source.

Fixed sources are normally used for those instances where values originate outside the analyzable object code. An additional optimization in our flow analysis allows the algorithm to use a fixed source in place of local sources for particular types. This is used, for example, for sources that represent exceptions in library methods since we rarely need to track any detailed information associated with these.

In addition to these various optimizations, the algorithm is designed to work with complete Java programs. This entails dealing with all the complications of such programs including native code, exceptions, threads, synchronization, callbacks, dynamic loading and binding using reflection, and large numbers of library routines. It also means tracking the implicit execution semantics of Java such as calls to static initializers and implicit field initializations. We handle the latter by encoding the implicit semantics into the flow analysis algorithms, for example ensuring that the static initializer for a class is called before we evaluate any methods of that class (unless the methods are called from within the static initializer). For the other issues, we use a common solution that lets the programmer specify routines that are to receive special handling.

Method special handling can take a variety of forms. Standard library and native methods can be flagged so that their internals are ignored and the value they return is a fixed source of the appropriate type. Methods that return values other than their declared type (e.g. are declared to return an interface type) can be declared to return a fixed source that is mutable. Such a source will be automatically converted to another fixed source upon an implicit or explicit cast. Other methods can be flagged with a substitute method. This is used for some internal java security calls, for methods that dynamically bind to implementation classes, and for methods such as Thread.start which actually invokes Thread.run asynchronously. Other methods, such as System.arraycopy need to be treated as special cases and are flagged as such. Finally, methods that register callbacks can be flagged so that the parameters for the callbacks will be computed correctly and the callback will be invoked as part of the dataflow.

The result is a package that does source-based data flow analysis of real java programs including all the various libraries and does it relatively efficiently. It handles small systems (5000 methods, 200,000 byte codes, 5000 in the system itself) in under 1 minute. On CHET itself (6400, 340,000/47,000), it takes under 3 minutes. On a larger project that includes 12 different executables (10158, 575,000/100,000) it takes 12 minutes.

Once this data flow analysis is complete, the framework identifies all instances of the specifications. It does this by finding, for each given specification, all instances of a source created by a trigger event for that specification. The instance is stored as the combination of the specification and this source.

6. Building Program Abstractions

Once we have identified an instance of a specification, we need to check that instance. We do this in two steps, first creating an abstraction of the application that only includes those portions that are relevant to the particular instance, and then checking if this abstraction meets the specification. The generated abstraction here is actually an abstract program that generates events for the specification.

6.1 The Abstract Program

This abstract program is generated to ensure that if there is an execution of the actual program which exhibits a certain sequence of specification events, then there is an execution of the abstract program that generates the same sequence. This is again 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 automata. 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.
- Exit a routine (return or end of program).
- Call a routine.
- Generate an event.
- Set a variable to a given value.
- Set the return value for the current routine.
- Test a variable or return value.

In addition, to facilitate checking of multithreaded applications, we have added the following actions:

- Asynchronous call of a routine.
- Begin synchronization for a set of sources.
- End synchronization for a set of sources.
- Wait or timed wait on a set of sources.
- Notify or notify all on a set of sources.

Execution of this abstract program is nondeterministic. Consider the single threaded case. At any point in time there is a current node. This node is executed to determine the current node at the next point in time. If the node is a call, then the current node is pushed onto the call stack and the next current node is the enter node of the called routine. If the node is a return, then the calling node is popped off the call stack. If there was no calling node, the program exits normally. If there was, then one of its successor nodes is chosen nondeterministically as the next node. If the node

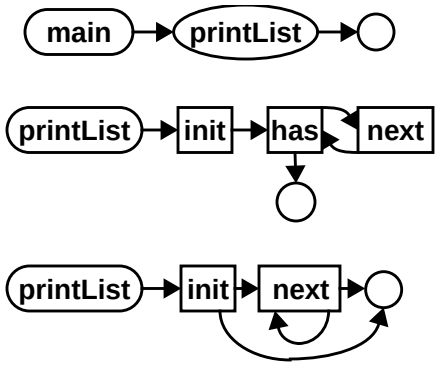

FIGURE 3. Generated automata for Simple.

is an event node, a variable set, or a return set, the next node is chosen nondeterministically from this node's successors after an event is output or the program state is changed accordingly. If the node is a test node and the test fails, the program fails; if the test succeeds then the next node is determined nondeterministically from the successors.

The threaded case assumes that there are multiple such programs with a common program state. A new thread is created by an asynchronous call node. Synchronization can be applied by keeping track of the set of sources that are currently being synchronized on and having a synchronize node block (i.e. use the current node as the next node) if synchronization would fail. This is not accurate in general since we cannot guarantee from the flow analysis that the same source from the analysis implies the same object at execution time; a true conservative approach requires us to ignore the synchronization statements. Wait and notify can also be modeled conservatively. A true wait blocks until there is a notify or notify all that shares a common source. Then it nondeterministically chooses to either continue blocking or to proceed to one of its successor nodes.

As an example of an abstract program, the code in Figure 2 yields the automata shown in Figure 3. Here entry nodes are rounded boxes containing the name of the routine, exit nodes are empty circles, and event generation nodes are boxes containing the event name. The automata for main is the same for both instances of the iterator check. The top automata is the one generated for the first instance of the iterator in *printList*, while the bottom automata is the one generated for the second instance.

6.2 Building the Abstraction

We build a program abstraction by mapping each method of the system being analyzed into an abstract program routine. The methods here are those that were used in the flow analysis phase, so that a method that was inlined in multiple versions actually appears as multiple methods in the abstraction. Moreover, we add additional methods to represent complex virtual calls, with the new method simply doing a parallel call of all possible alternatives as determined by the data flow analysis.

The program abstraction is generated in three phases. First, a prepass checks all the methods in the system to determine which ones are definitely not relevant to the given specification and instance. These methods and any calls to them can be ignored.

Next, we construct an automata for each potentially relevant method. This is done by making a symbolic execution pass over the code for the method. A new node is created whenever there is a synchronization entry or exit, when an event for the specification would be generated, when a method that is not ignored is called, when a monitored field is accessed, when a method returns, and at the start of each basic block. When a conditional occurs and one of the items tested is a monitored field or a return value, then nodes testing the value of the field are generated for the different resultant branches.

This generation is done using a path-sensitive analysis. While doing the symbolic execution, we keep track of the values on the stack and in local variables in a minimal way. For objects, we track whether the value is null or non-null. For numbers, we note if the value is constant and if so, what constant. When we have a branch in the program to the start of a basic block, we create a new abstract program node each time we have a different value set. This lets us construct finite programs that reflect local variable values. This value-based generation can be turned on or off for each particular method. We currently do it for all methods that are less than a certain size (currently 400 byte codes). This provides accuracy for most items while avoiding the relatively small number of cases where the procedure generates an initial excessively large graph.

The third phase of program abstraction is to simplify the resultant automata. This first involves local simplification, where we eliminate nodes without actions (e.g. all the extra nodes we inserted for basic blocks), eliminate empty or unneeded synchronized regions, eliminate unneeded tests, and eliminate meaningless returns. Second, it involves finding methods where the graph becomes trivial and eliminating these automata and any calls to them. These two steps are done concurrently using a worklist algorithm. A final step involves applying automata minimization to each remaining automata.

7. Checking Abstractions

Once we have generated the abstract program, the next step is to check whether all sequences of events that can be generated by that program are consistent with the given specification.

Many of the types of specifications we want to check. for example design patterns, UML interaction diagram, and class usage, are generally not thread-related. Thus, we first developed a means for efficiently and accurately checking specifications for the single threaded case and then extended this to the multithreaded case.

7.1 The Single Threaded Case

The overall approach to testing specifications is to determine the set of checking states that are reachable at each node of the program. We first define what we mean by a checking state. This has to reflect the program state and the state in the specification being checked. A *checking state* thus consists of a state from the specification (e.g. S2 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.

Next we determine for each routine and each possible checking state on entry to that routine, the set of checking states that are possible on exit. This is done using a worklist algorithm that takes a node and a set of states that can apply at the start of the node and then computes the set of states that apply at the start of any successors to the node. Each node is handled based on its associated action:

- Start nodes just propagate the current state to their successors.
- End nodes add the set of states that are generated to the states that apply after each corresponding call node, queuing up call nodes that might have changed.
- Call nodes check if the called method has been checked for each of the current states. If it has, they propagate the result states of the call to the successors; if not, they queue the starting node of the called method for later checking.
- Event nodes modify the state by applying the transition given in the specification for the given event.
- Field set and return nodes modify the state by changing the value of the field that is being set.
- Test nodes check the value of the field and either propagate the current state or nothing to their successors.

We note that this process handles recursion correctly. A recursive routine will be checked once for each achievable entry state. At least one of these states should represent the bottom of the recursion and thus should yield an output state. Propagating this state back, even through recursive calls, produces the correct set of output states in the light of recursion.

The final stage is to look at the possible exit states of each main program. These represent the final states that can be reached in any execution and thus indicate whether the specification succeeds by reaching an accepting state or fails by reaching an error state.

To handle programs that don't return or don't return if a particular state is reached, we distinguish specification states for which all transitions go to the state itself. These states typically 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 finds one final state for the first instance and two for the second. For the first instance, it notes that it is always the case that starting in main in state S1, one will end up in state S3. For the second instance, it finds that starting in state S1, one can actually end up in either state S2 where the iterator has been allocated but never used, or state S4, the error state.

7.2 The Multithreaded Case

We had several choices in extending this approach to handle real multithreaded programs. One approach would be to model each thread as a separate program as above and track the cross product of the states at each point. This would require, however, that the state include the call stack which would make it potentially infinite. The alternative we use is to find all instances of threads (based on asynchronous call nodes) and convert each into an automata based on the method graphs. This eliminates the call stack as part of the state while still preserving much of the information in the abstract program.

We build an initial thread automata using an inlining process, handling recursive calls by only having one copy of each method in the resultant graph. Then we simplify the resultant automaton first by removing unnecessary nodes and then doing automata minimization. The result is again a conservative approximation to the original program, ensuring that any execution of the thread in the original program will be reflected by some execution of the automata, but allowing executions of the automata that do not correspond to program executions.

Once we have constructed an automata for each possible thread in the program, we can extend the single-threaded checking approach to handle multiple threads. We start by extending the notion of a checking state to include thread information. We first add synchronization information to the checking state in the form of the set of sources that are currently synchronized for each active thread. Second, the checking state is extended to include the automaton node of each active thread. We allow a finite number of instances of each thread to be created, where the number is determined by the specification and defaults to three.

Next we extend the checking algorithm to deal with transitions caused by the threads. For each node, we augment the set of states at the start of the node with the set of all states that can arise by having any of the threads execute an action. This process is repeated to construct the full set of potential states for the given node.

Finally we handle the action nodes associated with threads, both when they are inside a thread automaton and when they are in the original program:

- For a begin synchronization node, we first check if the sources are in the current synchronization set for another thread. If they are and we are checking synchronizations then the resultant state set is empty. Otherwise we add the associated sources to the synchronization set for the current thread.
- For an end synchronization node, we remove the associated sources from the synchronization set.
- For a wait node, the set of states is empty unless we can execute a notify or notify all from that state on any of the same sources.
- For an asynchronous call node, we add a new thread instance to the current state provided that there not already too many instances of that thread currently active; otherwise the call is ignored.

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.

8. 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. While this is helpful, it is not enough information for a programmer to understand why or how the program can achieve 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 program rather than as an addition to the checking algorithm because the checking algorithm treats each method separately while the trace reporting has to keep track of the call stack and the actual sequence of method calls.

This pass uses much of the same techniques as the checking pass, doing a breadth-first search over the executions while tracking calls and attempting to find the specified target state.

The actual output from the framework consists of information gathered from each of the passes. Flow analysis reports all routines that are never used or that never return. For each instance of a specification, the output includes the specification, the source of the instance, and a top-level indicator of whether the particular instance passed or failed (or both). Then, based on the result of the data flow analysis, the output contains the program location (method, line, and even the instruction number) where each of the events that would be associated with the instance might be generated. This information could be used to instrument the application to dynamically check the instance if desired. Next, it outputs the program abstraction that was generated for the instance. Finally, it reports, for each different end state, the fact that there is an execution that yields that end state and includes a program trace for a sample such execution. All this is stored in an XML file that can either be read by the programmer or by another application.

Figure 4 shows excerpts from the output for the second instance of example of Figure 2. The first part of the figure identifies the specification and the source as well as indicating the status. The status OK-ERR here indicates that the specification can both succeed and fail. The next section of the output identifies the events that are used by this instance. Finally, the last portion of the output provides two program traces, one for each of the end states that was found. The actual output also includes the automata of the specification, the automata generated for each method for each instance, and more details on each of the event locations such as the instruction number.

9. Experience and Future Directions

We have been using this system to perform a variety of checks in a range of software systems. In particular, we have been checking the use of iterators and files in all systems. In addition, we have checked system-specific conditions such as the proper use of a support libraries for a web crawler and for a pinball program. CHET correctly identified several improper uses of iterators in our code (and in Sun's libraries), and pointed out files that were never closed. It also was instrumental in identifying odd cases in our web crawler sample program that were handled incorrectly. All this was done without any modifications or annotations in the systems being checked.

Performance has been dominated in most cases by the cost of flow analysis. For the largest system we have been working on (65,000 lines of source, 575,000 analyzed byte codes) [26], CHET identifies over 300 specification instances from within the project. The cost of generating and checking the resultant automata is less than one-tenth the cost of flow analysis, with most of the checks taking less than 10 ms., and only one requiring more that 100 ms. CHET itself comprises about 24,000 lines of Java code.

The system works as well as it does because of the detailed flow analysis and the use of simplified abstractions. The flow analysis lets us accurately identify instances of the specification without any user interaction.

```
 <TEST STATUS='OK-ERR' NAME='iterator_check' META='generic'>
<SOURCES>
      < SOURCE METHOD='spr.simple.Simple.printList' LINE='33' FILE='Simple.java' />
</SOURCES>
<EVENTS>
      <EVENTLOC STATEID='S_78' METHOD='spr.simple.Simple.printList' SIGNATURE='void()' FILE='Simple.
            <EVENT ID='E3' TRIGGER='false' NAME='next' TYPE='CALL' METHOD='next' />
      </EVENTLOC>
      <EVENTLOC STATEID='S_69' METHOD='spr.simple.Simple.printList' SIGNATURE='void()' FILE='Simple.
            <EVENT ID='E1' TRIGGER='true' NAME='NewIterator' TYPE='CALL' METHOD='iterator' />
      </EVENTLOC>
</EVENTS>
<CHECKER>
<TRANSITION START='S1' FINAL='S2' STATUS='OK'>
      <ELEMENTSTATE='S1'><EVENTLOC METHOD='spr.simple.Simple.main' FILE='Simple.java' LINE='9' /></ELEMENT>
      <ELEMENT STATE='S1' ><EVENTLOC METHOD='spr.simple.Simple.main' FILE='Simple.java' LINE='11' />/ELEMENT>
      <ELEMENT STATE='S1' ><EVENTLOC METHOD='spr.simple.Simple.printList' FILE='Simple.java' LINE='27' /></ELEMENT>
      <ELEMENT STATE='S1' ><EVENTLOC METHOD='spr.simple.Simple.printList' FILE='Simple.java' LINE='33' /></ELEMENT>
      <ELEMENT STATE='S2' ><EVENTLOC METHOD='spr.simple.Simple.printList' FILE='Simple.java' LINE='37' /></ELEMENT>
      <ELEMENT STATE='S2' ><EVENTLOC METHOD='spr.simple.Simple.main' FILE='Simple.java' LINE='12' /></ELEMENT>
</TRANSITION>
<TRANSITION START='S1' FINAL='S4' STATUS='ERR'>
      <ELEMENT STATE='S1' ><EVENTLOC METHOD='spr.simple.Simple.main' FILE='Simple.java' LINE='9' /></ELEMENT>
      <ELEMENT STATE='S1' ><EVENTLOC METHOD='spr.simple.Simple.main' FILE='Simple.java' LINE='11' /></ELEMENT>
      <ELEMENT STATE='S1' ><EVENTLOC METHOD='spr.simple.Simple.printList' FILE='Simple.java' LINE='27' /></ELEMENT>
      <ELEMENT STATE='S1' ><EVENTLOC S METHOD='spr.simple.Simple.printList' FILE='Simple.java' LINE='33'/></ELEMENT>
      <ELEMENT STATE='S2' ><EVENTLOC METHOD='spr.simple.Simple.printList' FILE='Simple.java' LINE='35'/></ELEMENT>
      <ELEMENT STATE='S4' ><EVENTLOC METHOD='spr.simple.Simple.printList' FILE='Simple.java' LINE='37' /></ELEMENT>
</TRANSITION>
</CHECKER>
</TEST>
```

FIGURE 4. Excerpts from the XML output for the simple sample program.

Moreover, it lets us simplify the abstract programs reflecting these specifications to the point where they are still accurate but easy to check.

Most of the work to date has concentrated on the single threaded cases. However, we have worked on checking multithreaded examples such as the web crawler cited above and a number of the thread-based specifications used as examples for Flavers and Bandera. Our experiences with the latter have been mixed. Where the program is relatively straightforward, we are able to do the checks and do them quickly, generally in under a minute. For some, such as the Flavers' dining philosophers example we get complete results. However, for others some of the checks are inconclusive because they require a more detailed analysis of program variables and data flow than our current defaults.

Our future work here will concentrate on making this system practical for checking a variety of conditions in large, complex systems. The particular directions we are interested in include:

- Finding easier ways of providing specifications to be checked and of extending our current specification language to handle more complex conditions such as nested objects and privacy concerns.
- Improving the performance and accuracy of flow analysis. One approach we are currently trying here is to model the basic library data structures directly rather than analyzing the library code.
- Incorporating thread information into the flow analysis so that thread-based specifications can be better written and checked.
- Making the generated automata more accurate by automatically detecting data fields that should be tracked as part of the analysis of a specification.
- Extending generation and checking to support limited arithmetic and a broader range of values on selected data fields.
- Extending generation to handle callbacks from native code (as in Swing). Such callbacks are handled correctly in the flow analysis, but currently are not incorporated into the abstract programs.
- Improving the checking of multithreaded applications either by an improved thread abstraction mechanism or by using a traditional model checker.
- Improving the user interface and robustness of the system so that we can make it available to students and other researchers and gain more experience.

Overall, however, we feel that our approach provides a practical means of checking dynamic specifications in real programs. The combination of data flow analysis to find instances of the specification along with the ability to create and check abstract programs rather than automata for each instance demonstrates a usable framework that has a lot of potential for use in future software systems.

10. Acknowledgements

This work was done with support from the National Science Foundation through grants ACI9982266, CCR9988141 and CCR9702188 and with the generous support of Sun Microsystems. Manos Renieris, Shriram

Krishnamurthi, and Philip Klein provided significant advise and feedback.

11. References

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

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

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

4. Guillaume Brat, 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).

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

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

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

8. James C. Corbett, 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).

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

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

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

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

13. Dawson Engler, 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).

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

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

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

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

18. Thomas A. Henzinger, Ranjit Jhala, Rpak Majumdar, and Kenneth L. McMillan, "Abstration from proofs," *Proc. POPL '04*, pp. 232-244 (2004).

19. Gerard Holzmann, *The Design and Validation of Computer Protocols*, Prentice Hall (1991).

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

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

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

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

24. I. Lee, 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).

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

26. Steven P. Reiss, "Constraining software evolution," *International Conference on Software Management*, pp. 162-171 (October 2002).

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

28. Willem Visser, 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).

29. Willem Visser, Klaus Havelund, Guillaume Brat, and Seung Joon Park, "Model checking programs," *IEEE Intl. Conf. on Automated Software Engineering*, (September 2000).