Specifying and Checking Component Usage

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

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. Unlike previous systems, CHET is able to check a wide range of complex conditions in large software systems without programmer intervention. This paper explores the specification techniques that are used in CHET and how they are able handle the types of specifications needed to accurately model and automatically identify component checks.

Categories and Subject Descriptors

D. Software; D.2 SOFTWARE ENGINEERING; D.2.4 Software/Program Verification; Subjects: Model checking; Programming by contract.

General Terms

Verification, Design, Experimentation, Languages.

Keywords

Components, verification, specifications, flow analysis, finite-state automata.

1. INTRODUCTION

Much of software engineering is concentrated on statically ensuring the reliability of software. Program-oriented work in this area includes safer languages, contracts, and tools for finding specific local 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 small software systems. While these efforts are helpful, they fail to address many of the problems of modern software development, especially problems related to the proper use of components.

1.1 Component Properties

Components are becoming more pervasive in software development. They exist in the form of large libraries such as those accompanying Java, remote object implementations such as CORBA, and web services. Components are typically more complex than traditional libraries, retaining state and requiring multiple calls of a variety of functions to perform these tasks. One of the major problems in dealing with components is ensuring that the application uses the components correctly.

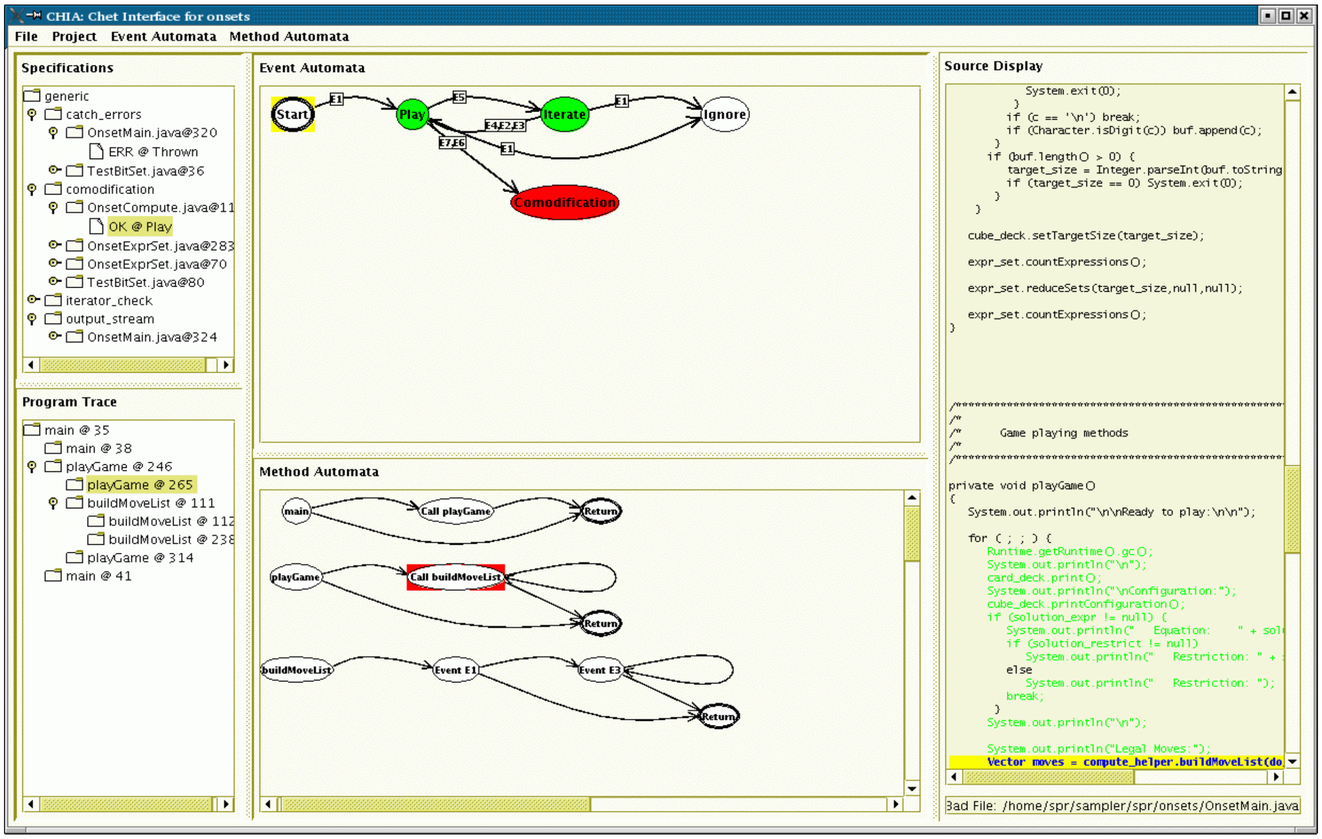
We wanted to develop a tool that would be able to check for the proper use of components in real software systems. Such a tool that would take a specification of how a component should be used along with an existing software system and would then identify possible executions of the program where the component was used incorrectly. While designing and implementing such a tool we realized that the same mechanisms could be used to check a wide variety of safety properties involving virtual components such as design patterns, internal components (e.g. user classes), and proper use of the programming language itself.

1.2 The CHET Architecture

To tackle this problem we have developed a system, CHET, that takes specifications and a user program and then produces a interactive display of where and how the various specifications are violated [28]. CHET works by splitting the task into distinct subproblems. The first problem is to provide a specification language that is suitable for describing component usage in an abstract way. These specifications have to be easy to provide, understandable by programmers, and general enough to describe component usage at a high level rather that describing each specific use of a component. For example, one should be able to specify that all instances of an XmlWriter library should obey certain properties without knowing what instances existed in the application or where they were used. CHET uses a specification based on extended automata over parameterized events to accomplish this. This is described in this paper.

The second problem involves finding, for each specification, all the particular instances that occur in an application. The identification of an instance is part of the specification problem. Finding all instances that meet the specification is done in CHET using a full interprocedural dataflow analysis that identifies both where instances occur and how they are used. For example, for the XmlWriter library, this would identify each location where an XmlWriter was created and also identify all uses of each creation so that each can be checked accurately.

The third problem involves creating a simple abstraction of the original program that accurately reflects a particular instance but that can be effectively checked. This is done in CHET by



```
              System.exit(0);
            }
          if (c == '\n') break;
          if (Character.isDigit(c)) buf.append(c);
        }
      if (buf.length() > 0) {
          target_size = Integer.parseInt(buf.toString
          if (target_size == 0) System.exit(0);
        }
    }

   cube_deck.setTargetSize(target_size);

   expr_set.countExpressions();

   expr_set.reduceSets(target_size,null,null);

   expr_set.countExpressions();
}




/*****************************************************************
/*
/*      Game playing methods
/*
/*****************************************************************

private void playGame()
{
   System.out.println("\n\nReady to play:\n\n");

   for ( ; ; ) {
      Runtime.getRuntime().gc();
      System.out.println("\n");
      card_deck.print();
      System.out.println("\nConfiguration:");
      cube_deck.printConfiguration();
      if (solution_expr != null) {
         System.out.println("   Equation:    " + sol
         if (solution_restrict != null)
            System.out.println("   Restriction: " + 
         else
            System.out.println("   Restriction: ");
         break;
       }
      System.out.println("\n");

      System.out.println("Legal Moves:");
      Vector moves = compute_helper.buildMoveList(do
```

Bad File: /home/spr/sampler/spr/onsets/OnsetMain.java

Figure 1. The CHET user interface

creating a nondeterministic abstract program containing calls, simple variables, tests, synchronization primitives, and output statements that generate events. To minimize the size of this program, CHET uses the results of the previous flow analysis and does additional flow analysis related to the particular case.

The final problem that CHET tackles involves comparing the abstract program with the original specification. Here CHET uses model checking techniques to simulate all possible executions of the abstract program and thus determine how the resultant output event sequences relate to the specification automaton.

CHET puts this all together with a user interface that lets the programmer browse over the different specifications and instances and then, having chosen one, look at the code and execution sequences that yield particular outcomes. An example of this is shown in Figure 1.

The remainder of this papers looks at the problems involved with developing a suitable specification languages. We start by describing the overall solution and then provide a series of examples that motivate the different aspects of the solution.

2. SPECIFICATIONS

To be practical our tool needs a specification model that is both easy to use and powerful. We want to check a wide variety of properties and we want to automatically identify each occurrence of those properties in real Java systems.

The underlying model we use for specifications is that of extended finite state automata over parameterized events. Finite automata are relatively easy to define for the various properties and are generally well understood by programmers. We use a modified form with bounded local variables to simplify nested specifications. The automata are driven by parameterized events that represent program actions or states. The set of available events is central to both specifying and identifying properties.

2.1 Events

Events represent the basic actions of the program relevant to a particular property. To express a broad range of properties, we need to have a variety of actions. For components, most of these actions involve method calls. 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. We also need to be able to restrict these events to a particular instance of the property. This is done by choosing the proper event set and having the events be parameterized. Parameters on events are used both to define and limit their occurrence. For example, an allocation event can define a parameter representing the object being allocated, while a call event can be limited to those cases where the *this* object represents the allocation. 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 [24]

The set of events that we currently provide include:

- CALL events 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, an argument of the method, or the *this* object of the calling method. Call events can restricted to be FLAT, i.e. to ignore anything that happens within the call.
- RETURN events triggered by a return from a particular method. This can be parameterized by the *this* object of the call or the value being returned.
- ENTRY events occurring when a method is entered, parameterized by the *this* object or an argument.
- FIELD events occurring 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 triggered by an allocation of an object of a particular class or subclass and parameterized by the object being allocated.
- CATCH events triggered by the execution of a catch clause. These are parameterized by the object that is the parameter of the catch.
- THROW events triggered when a throw clause executes and parameterized by the object being thrown.
- LOCK events triggered by the entry to either a synchronized method or a synchronized region. These are parameterized by the object being locked.
- UNLOCK events triggered by exiting a synchronized method or synchronized region, parameterized by the object being unlocked.

In each of these cases parameters can be used in two ways. Either they can be used to restrict the event by forcing the parameter to match existing values, or they can be used to define new values for the parameter. In the latter, we distinguish the case where the new value is exclusive (i.e. this *is* the defining instance of the parameter) and cases where the event adds to the set of values for the parameter. Parameters can also be restricted by the type of object, either explicitly or within the class hierarchy. All the events can be optionally restricted to only be considered in the project code rather than project and library code.

2.2 Event Parameters

CHET assigns specific events to program locations statically. This means that the parameters of the events are resolved before the abstract program is generated and all that CHET needs to consider in checking the abstract program are the resultant event sequences.

Determining which program locations can generate which events is done 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 in the program. 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. *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 are developed from the specification events.

Each specification must include one event that is marked as a *trigger*. This event must define at least one parameter and cannot check any parameters. Each instance of this event in the system identifies an occurrence of the corresponding property. During flow analysis, a new model source is created to represent the new parameters of all trigger events. The flow of this source then is used to determine what other events are relevant to the property. For example, an ALLOC trigger event identifies a new model source for each new statement for a particular class and hence an occurrence of the property for each such statement; the uses of the corresponding source would then identify the CALL or FIELD events that were relevant to the particular occurrence.

Specifications that define multiple parameters in separate events require special handling. First, we assume that the trigger event has to come first. Then if we locate a program location that is a candidate for a second parameter-generating event taking into account previously set parameter values, we use these to identify additional sources. These will generally be additional model sources created during flow analysis, but might simply be the set of local sources that flow to the location. These new source sets can then be used to identify additional events.

2.3 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 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. We assume that the automata is complete, i.e. that transitions are defined for each event for each state.

Transitions between the states consist of an event, a condition, and a set of actions. The condition and actions are based on named automata variables which can range over bounded integral values. Each variable has an initial value. Operations include *SET*, *INCR*, and *DECR*, and the set of tests includes checking if a value equals or doesn't equal a given value. 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.

A further extension using variables and conditions is to provide special values that represent the current (model) thread during simulation. Here we provide a *SETTHREAD* operator to set a variable to the representation for the current thread and a *TESTTHREAD* operator to check if the value of a variable matches that of the current thread. This extension lets us provide specifications where we can insure that two distinct paths occur in separate threads.

2.4 Specification Properties

Finally, each specification can provide directions on how it should be checked. In particular, it can specify information about threads, fields, exceptions, and synchronization. These properties are generally used to make the resultant tests more efficient.

Normally CHET determines for each instance of a property whether the events associated with the property can be guaranteed to all be generated within one thread. If so, it checks the program with thread starts being viewed as procedure calls; if not, it does a full multithreaded check. Since the programmer isn't worried about threaded dependencies for some properties, CHET lets the programmer specify that a check should be treated as nonthreaded.

Second, CHET normally determines the set of fields used in checking each instance of a condition using a separate flow analysis pass and a set of heuristics. It some cases, especially for application-specific properties, the programmer might know a priori which fields are relevant. CHET lets the program both specify particular fields to be considered and to disable the automatic detection of fields.

Third, one needs to consider what exceptions to consider in doing the checks. One obviously wants to consider all exceptions which are either explicitly thrown or (for native or library routines) explicitly declared to be thrown. The question that comes up however, is whether you should consider other possible exceptions or errors that are caught by the code but never explicitly thrown. This case arises more frequently than one would think since each synchronized region internally catches and then rethrows *Throwable* to ensure the corresponding monitor exits cleanly. CHET lets the programmer specify whether or not to consider such unusual exceptions while checking.

Finally, CHET has the ability to model synchronized regions during checking, insuring that two model threads don't execute synchronized code at the same time. This check is only done if CHET determines that synchronization might be relevant to the checking. CHET's determination here is conservative and the programmer might have a better understanding of the issues. Since keeping track of synchronization information during checking can be expensive, CHET lets the programmer disable this check for a particular property.

3. EXAMPLES

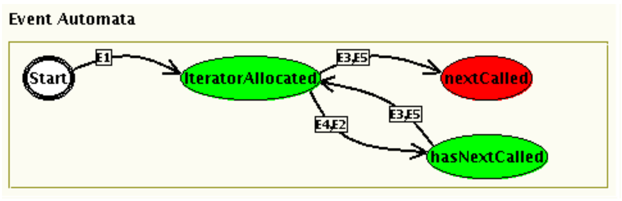
In this section we look at a variety of specifications and use them to illustrate the power and flexibility of CHET's specification language.

3.1 Iterators

We first consider checking the proper use of iterators. Here we want to ensure for each use of an iterator in the program, that we always call *hasNext* before we call *next*. Similar checks can be made for enumerations, string tokenizers, and stream tokenizers.

The automaton for this check as shown in Figure 2 is fairly simple. Here we show the automaton as it is displayed by our interactive tool and the set of events that are associated with the automaton. (In the tool, event and variable information are shown through tool tips.) Self loops are not displayed for clarity. Note that the parameter C1 is defined by the initial event and then simply used by the remaining events.

The automaton starts with the trigger event E1 which represents a new instance of an iterator. We originally used an ALLOC event here, but found, by looking at the resultant reports, that this did not do the appropriate checking since all iterators for each particular type of data structure (e.g. all Vector iterators) are allocated at one point in the code. Instead, we identify an instance of the specification with a return from the method *Collection.iterator()*. In this state if we get a call to *next* (or *nextElement*) we go to the error state. If we get a call



Event	Type	Parameter
E1 (New)	RETURN(iterator)	C1 = result
E2 (HasNext)	CALL(hasNext)	this == C1
E3 (Next)	CALL(next)	this == C1
E4 (HasMore)	CALL(hasMoreElements)	this == C1
E5 (NextElt)	CALL(nextElement)	this == C1

Figure 2. The Iterator Specification

to *hasNext* (or *hasMoreElements*) we go to the *hasNextCalled* state. Here, we can get a call to *next* and go back to the *IteratorAllocated* state.

Since CHET does partial context-sensitive flow analysis, we could have left the trigger event be the allocation and told the system to view the various Collection methods as context sensitive. This would have been closer to what we did, but would have treated all iterators for a given collection object as a single instance.

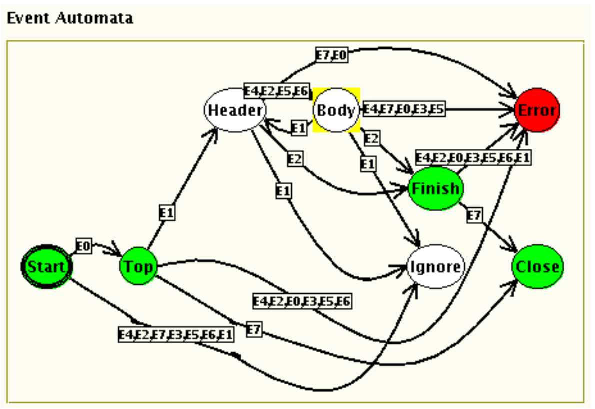
We also note that CHET, because it does full flow analysis, checks iterators whether they are defined and used in a single routine or a defined, passed around, and used over multiple routines and multiple calls.

3.2 An XML Writer class

We next consider a slightly more complex example where we are specifying the behavior of an XML writer class. This class provides methods to begin and end the output of an XML element. These methods can be nested to provide for subelements. In addition, the class provides calls to set attributes of the current element and to add text to the current element. Usage of the class is restricted so that all attributes must be added before either text or subelements are added and the begin and end pairs must match up. The automaton and event table for the corresponding specification are shown in Figure 3. Again self-loops are elided from the diagram. In addition, the use of variables and conditions is not shown directly.

The trigger for this specification is the allocation of an *XmlWriter* element (E0). This puts the automaton into the *Top* state. At this point we expect a call to *begin* (E1), which puts the automaton into the *Header* state. Here attributes can be created at will (using the *field* method, E3). A call to add text or end a subelement puts the automaton into the *Body* state where attempts to add additional attributes or text are errors. Starting a new subelement puts the automaton back into the *Header* state. Finally, ending the top-level element puts the automaton into the *Finish* state. A call to *close* in the middle of writing an element is an error. The *Ignore* state is there to trap additional allocations that would indicate false paths in the program.

This automaton needs to keep track of the level of nesting of elements to perform correctly. This is done by adding a vari-



Event	Type	Parameter
E0(New)	ALLOC(XmlWriter)	C1 = new
E1(Begin)	CALL(begin)	this == C1
E2(End)	CALL(end)	this == C1
E3(Field)	CALL(field)	this == C1
E4(Cdata)	CALL(cdata)	this == C1
E5(Text)	CALL(text)	this == C1
E6(Xml)	CALL(writeXml)	this == C1
E7(Close)	CALL(close)	this == C1

Figure 3. Xml Writer Specification

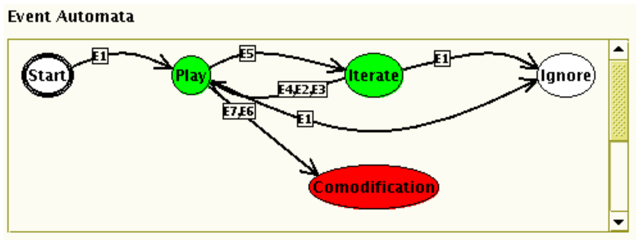
able, *lvl*, to the automaton. The variable initially is assigned the value 0. A transition on E1 from the state *Top*, *Header*, or *Body* increments *lvl* by 1 unless the value is already greater than a preset limit (10), in which case the transition is to the *Ignore* state. CHET will then check all paths through the code which have a nesting level less than this limit. A transition for E2 decrements the variable if it is greater than 0. A transition from E2 with a level of 1 goes to state *Finish*.

3.3 Comodification

The above examples use one parameter to define the events. Checking for potential instance of comodification in Java, i.e. places where the code modifies a structure at the same time it is iterating over the structure (which causes a *Comodification-Exception* to be thrown), requires two parameters, one for the original data structure and one for the iterator. The automaton and event table for this check is shown in Figure 4.

Here the *Start* state represents the beginning of the program. An allocation of a *Vector* is the trigger event E1; it causes us to enter the *Play* state. We can modify the vector in this case but if we try to iterate, we get an error. Calling the *iterator* method on the vector yields an *Iterator* object, parameter C2, and puts us in the *Iterate* state. Here we can call.*next*, but any attempt to change the vector causes us to go back to the *Play* state. The *Ignore* state handles cases where a new vector replaces the original. An attempt to use the iterator in the *Play* state indicates a potential comodification error and cause the automaton to enter the *Error* state.

One complication that arises here is that there can be many iterators created for a given Vector. CHET lets us check each



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

Figure 4. Comodification Specification

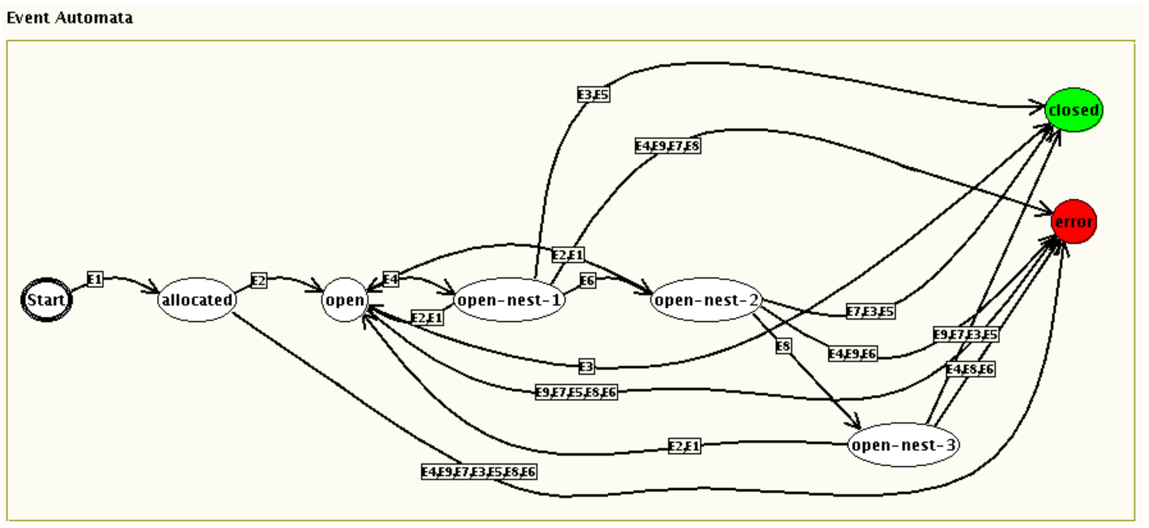
one separately or together. For this automaton, we set the match type for the C2 parameter on event E5 so that new values are accumulated and all uses are checked with one specification. To check each instance separately, we would create a slightly different automaton. Here the trigger event would be the creation of the iterator and it would define both parameters, C1 from the *this* value and C2 from the return value. Similar automata can be provided for checking comodification for other data structures.

3.4 File Open and Close

The next specification checks that all files that are opened by the user are explicitly closed. While this can be thought of as an easy specification to write, the simplest form, i.e. that a call that creates a new *FileWriter* must be followed by a call that closes that file, is tricky to check and generally not true given the current Java libraries. Doing an accurate and complete check requires a more sophisticated automaton.

The reason the simple check doesn't work is inherent in the way files are used in Java. Typically the program will create an instance of a *FileWriter* and will then use it to create an instance of a *BufferedWriter* which will in turn be used to create an instance of a *PrintWriter*. When the program is done, it closes the *PrintWriter*. While CHET is powerful enough to determine that closing the *PrintWriter* should eventually close the *FileWriter*, these checks require knowledge of internal variables and modeling the internals of the Java I/O libraries. Moreover, if the *PrintWriter* gets an error flushing the file on the close call, the underlying file is actually not closed — a potential error in the library, but not the user's code.

The actual automaton we check models the nested use of files explicitly. It is shown in Figure 5. This automaton supports a fixed level (3) of nesting that is sufficient for most programs. It explicitly models the nested writers and checks for a close of any of the nested elements. The trigger event is an allocation of a *FileWriter*. The automaton first needs to check if the file is successfully opened. This is done by checking for a successful



Event	Type	Parameter
E1(Create)	ALLOC(FileWriter)	C1 = new
E2(Open)	RETURN(<init>)	this == C1
E3(Close)	CALL(close)	this == C1
E4(Nest)	CALL(<init>	arg1 == C1 C2 = this
E5(CloseNest)	CALL(close)	this == C2
E6(Nest1)	CALL(<init>)	arg1 == C2 C3 = this
E7(CloseNest1)	CALL(close)	this == C3
E8(Nest2)	CALL(<init>)	arg1 == C3 C4 = this
E9(CloseNest2)	CALL(close)	this == C4

Figure 5. Specification to check if files that are opened successfully are closed.

return from the constructor (*<init>* denotes a constructor here). After this, the automaton enters state *open* and checks for a close. If a successful nested open of a subclass of *Writer* with the given file as the first argument occurs, it enters the corresponding *open-nest* state and checks for a close of either the original or the nested writer. Note that by handling subclasses, we are also able to handle user-defined *Writer* classes.

To make the checking of this more efficient, we want to tell CHET, as part of the specification, that it only needs to consider the application program and not the underlying libraries. This is done in two ways. First, the initial close event is marked as *GLOBAL* indicating that it should only consider calls in the user's code. Second, the various nested close events are marked as *FLAT* indicating that CHET should ignore the internals of those calls when doing the checking.

Similar automaton can be created for objects of class *FileOutputStream*, *FileInputStream*, and *FileReader*.

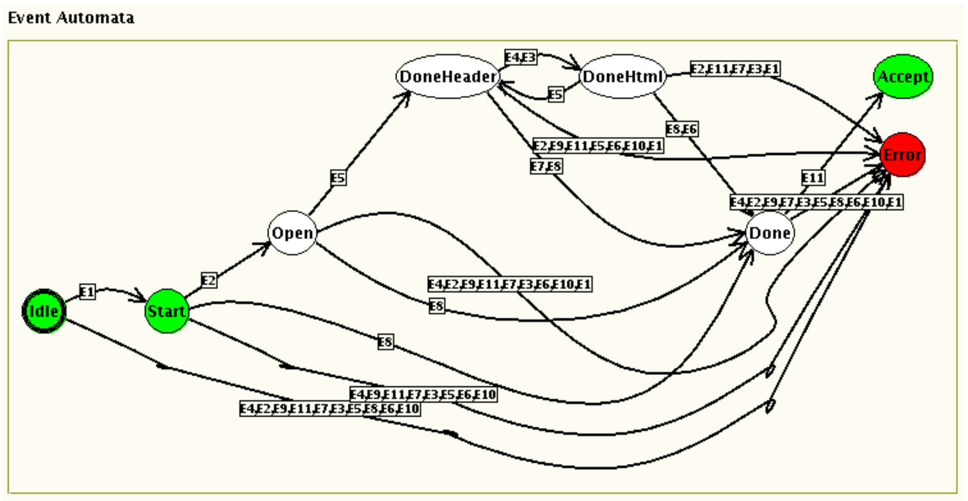
3.5 Web Crawler Library

One of the our course assignments is to create a multithreaded web crawler. The support code for the assignment includes a library that stores information about the resultant pages and determine whether pages have been previously scanned or not. The library has to be used in a particular way. Once the user code gets a URL to potentially scan, it needs to call a function, *beginProcessing*, that checks if the URL has already been scanned and, if not, ensures that this user has exclusive rights to scan it. Then the user code needs to actually open the URL connection. If the open fails, the code needs to call an routine to flag an error; if the open succeeds but indicates redirection, the code needs to call a redirection routine. Finally, if the open succeeds, then the code should save the HTML either directly or indirectly, and then as it parses the page, call a succession of routines that indicate text fragments and text breaks and then, when done, save the set of links on the page. In all cases the *endProcessing* routine must be called when processing of the URL is complete. Needless to say, many students do not initially use the library correctly.

A CHET specification that does much of the checking for correct usage is shown in Figure 6. This specification does not check everything that is required however. In particular, it does not take into account the status that is returned on opening a connection and ensuring that the proper paths are taken based on the status value. Because there are different ways for a user to determine the error code and the code is actually determined not in the application but from data that is read, this is quite difficult to specify without looking at the application code. However, the check that is here is still sufficient to catch many of the student misuses of the library.

3.6 Checking thrown errors

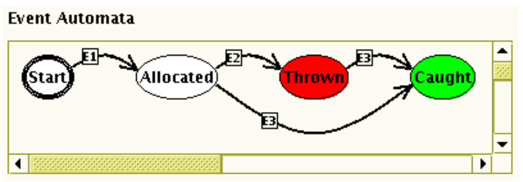
CHET and its specifications are useful beyond just checking for proper use of components. For example, exceptions in Java are complicated because once they are declared for a method they have to be caught by each caller. Programmers sometimes avoid having to provide the extra code by throwing errors rather than exceptions. While this is simpler for the program-



Event	Type	Parameter
E1(Begin)	CALL(beginProcessing)	C1 = arg1
E2(Open)	RETURN(openConnection)	this == C1
E3(Save)	CALL(saveHtml)	this == C1
E4(File)	CALL(getHtmlFile)	this == C1
E5(Header)	CALL(saveHeader)	this == C1
E6(Links)	CALL(saveLinks)	this == C1

Event	Type	Parameter
E7(Redirect)	Call(setRedirectHtml)	this == C1
E8(NoteErr)	CALL(setError)	this == C1
E9(Text)	CALL(processText)	this == C1
E10(TextBrk)	CALL(processTextBreak)	this == C1
E11(Finish)	CALL(endProcessing)	this == C1

Figure 6. Specification to check for proper use of web crawler library.



Event	Type	Parameter
E1(Create)	ALLOC(Error)	C1 = new
E2(Throw)	THROW	throw == C1
E3(Catch)	CATCH	catch == C1

Figure 7. Catching Errors Specification

mer, it is a bad programming practice and should generally not be done. A CHET specification that checks for this is shown in Figure 7.

This specification is simple. The trigger event is any allocation of an instance of *java.lang.Error* in the user's application. If the resultant object is then thrown, the automaton goes into the *Thrown* state. The error must then be caught explicitly in the user's code to move the automaton to the *Caught* state. If the program can exit with the automaton in the *Thrown* state, the specification indicates an error.

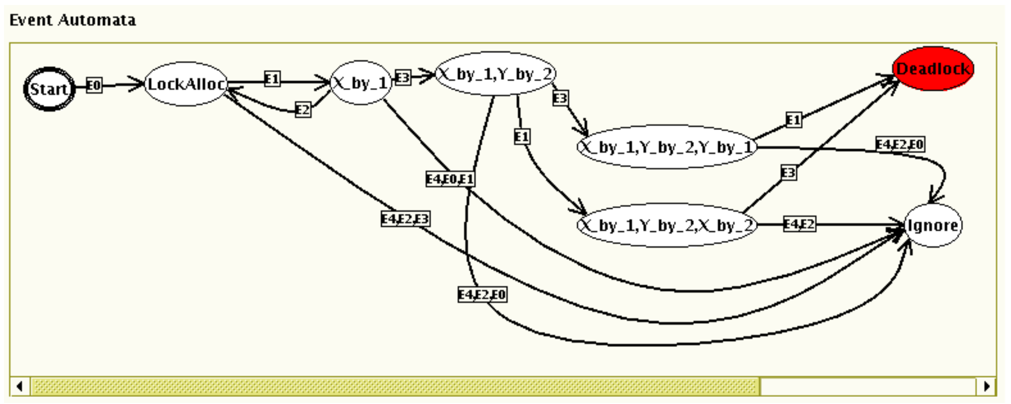
3.7 Checking for Deadlock

Another example of language checks is checking for deadlock. While CHET isn't designed as a general mechanism for checking for deadlock and while deadlock can occur in many forms, it is possible to create specifications that check for particular types of deadlock. For example, the specification of Figure 8 checks for two threads locking synchronized regions in different orders, i.e. the first locks A and then B while the second locks B and then A.

The set of events here is deceptively small since the automaton actually needs to detect different locks in different threads. The automaton specifies this internally using two automaton variables, *t1* and *t2*, that represent threads. The value of *t1* is set to the current thread when the first lock event (E1) occurs. The second variable, *t2*, is set the first time the second lock is set. The condition here requires that *t1* be distinct from *t2*. Finally, the remain transitions are restricted by conditions stating which thread they occur in. Note that the event E3 specifies that the value for C2 must be distinct (unique) from other parameters, i.e. the lock represented here must be different from the lock represented by C1. This ensures that multiple locks are being considered.

4. 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 [29], static condition checking as in CCEL [11], and verification-based static checking such as in LCLint [15]. More recently and more related to



Event	Type	Parameter
E0 (Alloc)	ALLOC(Object)	C1 = new
E1 (Lock_X)	LOCK	lock == C1
E2 (Unlock_X)	UNLOCK	lock == C1
E3 (Lock_Y)	LOCK	C2 = new (U)
E4 (Unlock_Y)	UNLOCK	lock = C2

Figure 8. A specification checking for potential deadlock based on lock order.

our work, there has been a significant body of work on software model checking [21].

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. CHET combines and extends aspects of a number of other systems to meet the requirements outlined above.

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 [8,9,12] and Flavers [7] 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 [14] and ESP [10] 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 [25] 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 [13]. MAGIC uses labeled transition systems (LTS) to model procedures where the labels correspond to program statements [6]. Java Pathfinder [17,18] uses linear temporal logic (LTL) which is a more compact representation related to finite automata but more removed from the programming language and the programmer. Java Pathfinder 2 [4,26,31] requires that the user create the necessary conditions as Java code that is then analyzed with the application. Easl uses Java code fragments that define behaviors [27]. The Alloy system has its own set-theoretic language for defining specifications [23]. 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 [30] 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 [22]. 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 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 [19]. Later work on BLAST uses Craig interpolation and proof techniques to better the abstraction [20]. 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

System	LOC	#BC	Proj #BC	flow time	# Tests	# Warnings	Avg Test	Max Test	Total Time
Onsets	2669	155645	6248	44.97	10	3	3.4	17.0	1:05.91
Crawler	3556	205514	5455	68.14	25	11	3.5	14.0	1:25.11
Pinball	11264	241264	54892	72.89	39	7	61.8	673.0	1:38.63
Freecs	20570	228163	50567	109.77	68	2	10.7	452.0	3:05.62
Taiga	51391	194923	15404	52.63	39	3	2.36	44.0	1:33.96
Egothor	54317	559898	268663	371.89	152	27	5.7	66.0	9:46.06
Clime	64998	421709	117575	255.85	571	20	14.0	2563.0	13:06.20
Jalopy	94636	639310	348843	1697.79	36	9	18.5	222.0	31:32.11
Openjms	95470	308198	30787	384.53	54	45	2.24	30	7:19.74

Figure 9. The results of running CHET on various systems.

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 [4,26]. 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.

5. EXPERIENCE

We have used CHET successfully for a variety of different specifications and a variety of different systems.

In addition to the specifications outlined here, we have looked at specifications that check whether Swing objects have the appropriate callbacks registered, multiple checks to ensure that a Java byte code library is used correctly, checks for proper use of the JoGL library. and checks for several design patterns including Singleton and Chain of Responsibility [16].

The table in Figure 9 summarizes some of our experience with CHET. The first column 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 indicate the size of the systems, first in lines of code, then in the total number of byte codes analyzed and the number of byte codes analyzed from within the project (i.e. the user's code). 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. The next two columns indicate the number of test instances that were identified and the number of errors that were detected. Discrepancies here are generally due to the fact that while we correctly identify and do flow analysis for specification instances that occur during callbacks, we do not yet test such instances. The next two columns indicate the average and maximum time for a single test in milliseconds. The final column is the total time taken by CHET to analyze the system.

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 table shows that the flow analysis essentially dominates the performance of the system and the flow analysis is dependent mainly on the complexity 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.

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

Acknowledgements

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.

6. 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. Sagur Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith, "Modular verification of software components in C," *IEEE Trans. on Software Engineering* Vol. **30**(6) pp. 388-402 (June 2004).
7. 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).
8. 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).
9. 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).
10. Manuvir Das, Sorin Lerner, and Mark Seigle, "ESP: Path-sensitive program verification in polynomial time," *Proc. PLDI 2002*, (June 2002).
11. Carolyn K. Duby, Scott Meyers, and Steven P. Reiss, "CCEL: a metalanguage for C++," *Proc. Second Usenix C++ Conference*, (August 1992).
12. 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).
13. 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).
14. 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).
15. 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).
16. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley (1995).
17. 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).
18. 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).
19. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre, "Lazy abstraction," *Proc. POPL '02*, pp. 58-70 (2002).
20. Thomas A. Henzinger, Ranjit Jhala, Rpak Majumdar, and Kenneth L. McMillan, "Abstraction from proofs," *Proc. POPL '04*, pp. 232-244 (2004).
21. Gerard J. Holzmann and Margaret H. Smith, "Software model checking," *Forte*, pp. 481-497 (1999).
22. 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).
23. Daniel Jackson and Alan Fekete, "Lightweight analysis of object interactions," *Proc TACS 2001, Springer-Verlag LNCS 2215*, pp. 492-513 (2001).
24. 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).
25. 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).
26. 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).
27. G. Ramalingam, Alex Warshavsky, John Field, Deepak Goyal, and Mooly Sagiv, "Deriving specialized program analyses for certifying component-client conformance," *PDLI 2002*, pp. 83-94 (June 2002).
28. Steven P. Reiss, "Checking event-based specifications in Java systems," *Proc. SoftMC 2005*, (July 2005).
29. 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).
30. 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).
31. Willem Visser, Klaus Havelund, Guillaume Brat, and Seung Joon Park, "Model checking programs," *Automated Software Engineering Journal* Vol. **10**(2)(April 2003).