# Tool Demonstration: CHET: Checking Specifications in Java Systems

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

## Abstract

*One of today's challenges is understanding the behavior of complex software. A major challenge here is that libraries, classes, and other components are often not well understood and can easily be used incorrectly. Our system, CHET, lets developers define specifications that describe how a component should be used and checks these specifications statically 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. CHET comes with an interactive front end that makes both defining specifications and understanding the results of the checks intuitive and easy.*

## 1. Introduction

Much of software engineering is concentrated on ensuring the reliability of software. Work in this area includes safer languages, contracts, and tools for finding specific 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 software. While these efforts are helpful, they fail to address many of the problems of modern software development and understanding.

Our system can check a wide range of software properties dealing with the use of libraries and components. Examples of the properties we can specify and check include:

- For each use of an iterator in a program, *hasNext*() is called exactly once before *next*() is called.
- Iterators are not used while the structure is being modified outside of the iterator (concurrent modification).
- Each file opened by the application is closed.
- Each class that represents an instance of a singleton design pattern [13] has only one associated object.
- User defined errors (subclasses of *java.lang.Error*) thrown by the application are caught.
- An XML writer component is used such that begin and end pairs match up and fields are added before any subelements.
- For a Java byte code library component, a current function is registered before any attempts to find a line number for an instruction are made.
- For a web crawler, each page that is found is processed correctly by either indicating error, providing a redirected URL, or by providing the page contents and the internal text to a URL support library.
- An object of class A is always locked before an object of class B.
- A session id is successfully obtained from a web service before the action methods are called.

In order for a tool that addresses properties of this sort to be successful it must meet certain requirements. First, the properties must be easy to specify. Second, the tool needs to check separately each instance of the property in the program, for example each use of an iterator. Third, the identification of instances has to be automatic. Fourth, the tool has to be fast enough so that programmers can use it everyday. Finally, the tool had to be as accurate as possible. It has to be sound in that it correctly identifies each instance of each property in the program that does not meet the specification, and it has to minimize false positives.

Our system, CHET, meets these requirements. The system has been used to check the above and other properties on a variety of software systems that include simple test programs, small (1-3 Kloc) student programs from a software engineering course, our current programming system CLIME (of which CHET is a part; a total of about 68 Kloc), and various open source projects (egothor, j-ftp, jalopy, openjms, heretrix) ranging from 25 Kloc to 94 Kloc. These test have shown the system is accurate and robust and scales for large systems.

Performance is quite acceptable. For the CLIME system (68,000 lines of source, 370,000 analyzed Java byte codes), CHET takes about 12 minutes to identify over 540 specification instances from within the project and check each of these instances individually. Most of the smaller systems are checked in under a minute. The system is available



Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)
1092-8138/05 $ 20.00 **IEEE**

online at *http://www.cs.brown.edu/people/spr/research/chet.html*.

## 2. The CHET System

CHET is based on a specification model that is broad enough to deal with a wide variety of properties. The model uses 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 an extended form that uses bounded local variables and Boolean conditions on the arcs to simplify nested specifications. The automata are driven by events representing program actions or states. The set of available events is central to both specifying and identifying properties.

Events represent the basic actions of the program relevant to a particular property. In order to express a broad range of properties, we provide a variety of events. For components, most of these events relate to calling methods of the components. Other relevant events include object allocation, setting fields, handling exceptions, and synchronization.

To support automatic identification of property occurrences, we need to be able to statically detect the events describing what is going on at run time. More importantly, we must be able to restrict these events to a *particular occurrence of a property.* This is done by having the events be parameterized. Parameters on events are used both to define and limit their occurrence. For example, we can look at the use of a particular *Iterator* in the program by associating a parameter with a particular allocation of the iterator and then only looking at the calls to the methods *next* and *hasNext* that can be have that particular object as the *this* parameter.

Identifying events and determining possible values for the associated parameters is accomplished in CHET through a full interprocedural flow analysis of the program and its libraries. This analysis takes into account callbacks, native methods, reflection, and other complexities of Java systems. Its output is used to find the possible locations for all events of each specification, to identify all instances of each specification, and to ensure the correct parameterization of events for each such instance.

To efficiently check each occurrence of a specification in the user's application, CHET generates an abstract program that models the application's behavior with respect to that particular occurrence. The abstract program is restricted to include only a simplification of that part of the code that can generate events relevant to the occurrence. This abstraction ensures that if there is an execution of the actual program exhibiting a certain sequence of specification events, then there is an execution of the abstract program generating the same sequence. This is conservative in that the abstract program may generate sequences that can never be exhibited in the actual program.

The abstract program consists of a set of routines. Each routine is composed of nodes and arcs similar to an automaton. There are actions associated with each node, but the arcs are uninterpreted. The associated actions control the behavior of the program and the generation of events. Typical actions include calling a routine, setting a variable, generating an event, returning, and condition checks.

The goal of CHET is to ensure that in all possible executions of the original program each occurrence of a specification must obey the specification rules. Checking that this actually occurs is done by considering all possible executions of the abstract program and seeing what specification states can result.

CHET does this using model checking techniques. For each possible execution it determines what specification states can occur at each point in the abstract program. It does a detailed single-threaded analysis in those cases where it can determine that only one thread can affect the specification. Here it looks at each method in the abstract program and determines for each possible input specification state, what specification states are possible on exit. For the multithreaded case, it first identifies all potential threads and then collapses the abstract program representing each thread into a single automaton by inlining the calls. It then runs the various thread automata in parallel with the main program.

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

To make the output easy to understand, we provide an interface application that presents the results of the analysis to the user in a meaningful way. Examples of the interface are shown in Figure 1. The display is split into five parts. The pane at the upper left lets the user select which specification to look and then what instance of that specification using a tree view. Once an

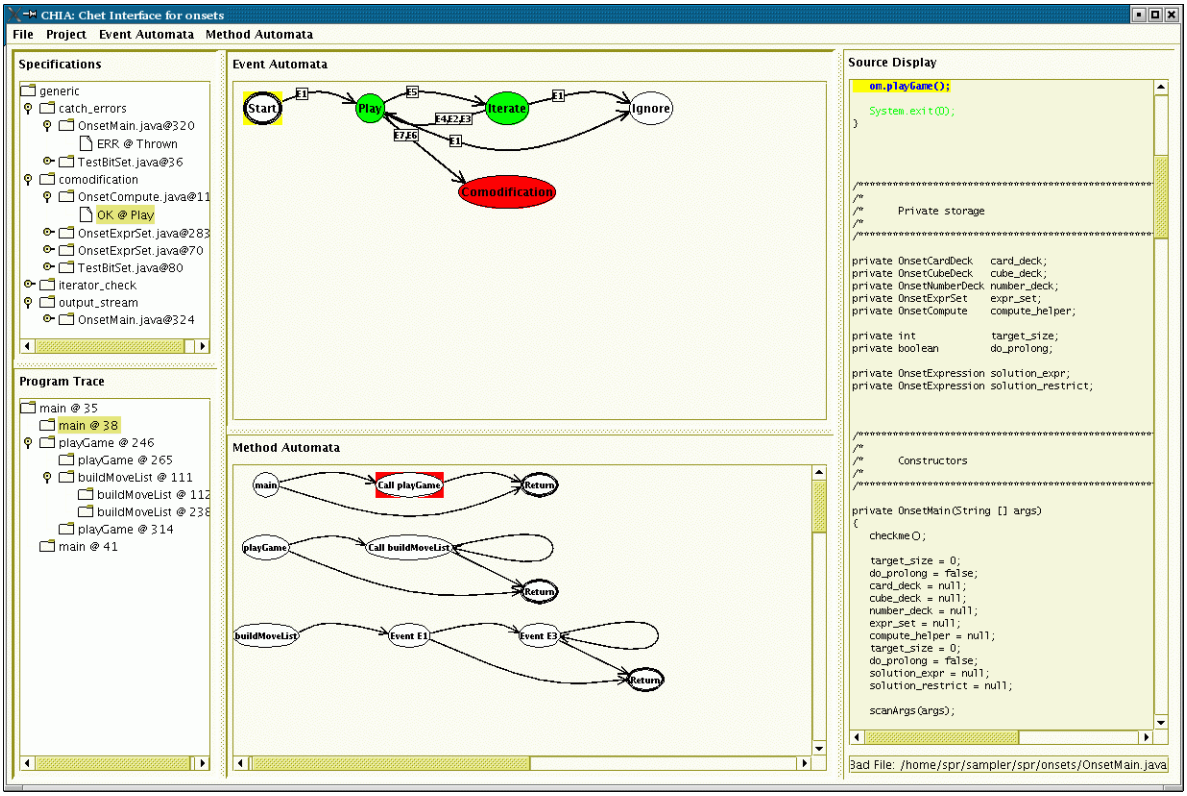


**FIGURE 1. Chet interface display showing a comodification check.**

instance is selected, the two center panes show first the automata representing the specification and then the abstract program that is generated from the original source for this instance. A example or counterexample (depending on whether checking found an error or not) is shown as a call tree at the bottom left. As the user walks through this tree, the appropriate nodes are highlighted in the two automata and the corresponding source is displayed in the panel to the right.

## 3. 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 [20], static condition checking as in CCEL [8], and verification-based static checking such as in LCLint [12]. More recently and more related to our work, there has been a significant body of work on software model checking [1-7,9-11,14-19,21].

Our work is most closely related to the work on model checking. It differs in that it provides an easy-to-use automata-based model for defining abstract properties to be checked, then automatically finds all instances of those properties in the program, and then model checks each instance separately. It also differs in its ability to handle relatively large programs. In doing this in combines techniques from several existing systems. Its use of finite automata builds on that of Bandera and Flavors which use explicit events rather than CHET's implicit ones. It analyzes bytes codes in order to deal with full systems as do Java Pathfinder 2 and Bandera. It generates a simple abstract program that similar to that of Flavors but that includes more conditions and function calls. It uses a simple but effective context-free model checker at the back end, combining this with an inlining technique similar to Flavors in the multithreaded case.

## 4. Conclusion

CHET demonstrates that it is possible to effectively and automatically check significant properties in large software systems. Specifications consisting of simple automata over parameterized events let us isolate check specific instances of a variety of properties. Our checking techniques provide a high degree of accuracy and useful results.

## 5. 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. 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).

3. 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).

4. 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).

5. 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).

6. 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).

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

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

9. 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).

10. 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).

11. 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).

12. 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).

13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley (1995).

14. 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).

15. 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).

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

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

18. 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).

19. 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).

20. 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).

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