A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs

Sagar J. Chaki

January 26, 2005, 9:30 AM – 11:30 AM
Newell Simon Hall, Room #1507
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis Committee:
Prof. Edmund M. Clarke, CMU, Chair
Prof. Randal E. Bryant, CMU
Prof. David Garlan, CMU
Dr. Sriram K. Rajamani, Microsoft Research

The ability to reason about the correctness of programs is no longer a subject of primarily academic interest. With each passing day the complexity of software artifacts being produced and employed is increasing dramatically. There is hardly any aspect of our day-to-day lives where software agents do not play an often silent yet crucial role. The fact that many of such roles are safety-critical mandates that these software artifacts be validated rigorously before deployment. So far, however, this goal has largely eluded us.

In this article we will first layout the problem space which is of concern to my thesis, viz., automated formal verification of concurrent programs. We will present the core issues and problems, as well as the major paradigms and techniques that have emerged in our search for effective solutions. We will highlight the important hurdles that remain to be scaled. The later portion of this article presents an overview of the major techniques proposed in my thesis to surmount these hurdles. The article ends with a summary of the core contributions of my dissertation.

Software Complexity

Several factors hinder our ability to reason about non-trivial concurrent programs in an automated manner. First, the sheer complexity of software. Binaries obtained from hundreds of thousands of lines of source code are routinely executed. The source code is written in languages ranging from C/C++/Java to ML/Ocaml. These languages differ not only in their flavor (imperative, functional) but also in their constructs (procedures, objects, pattern-matching, dynamic memory allocation, garbage collection), semantics (loose, rigorous) and so on.

This *sequential* complexity is but one face of the coin. Matters are further exacerbated by what can be called *parallel* complexity. State of the art software agents rarely operate in isolation. Usually they communicate and cooperate with other agents while performing their tasks. With the advent of the Internet, and the advance in networking technology, the scope of such communication could range from multiple threads communicating via shared memory on the same computer to servers and clients communicating via SSL channels across the Atlantic. Verifying the correctness of such complex behavior is a daunting challenge.

Software Development

Another, much less visible yet important, factor is the development process employed in the production of most software and the role played by validation and testing methodologies in such processes. A typical instance of a software development cycle consists of five phases - (i) requirement specification, (ii) design, (iii) design validation, (iv) implementation and (v) implementation validation. The idea is that defects found in the design (in phase iii) are used to improve the design and those found in the implementation (in phase v) are used to improve the implementation. The cycle is repeated until each stage concludes successfully.

Usually the design is described using a formal notation like UML. The dynamic behavior is often described using Statecharts (or some variant of it). The design validation is done by some exhaustive technique (like model checking). However, what matters in the final tally is not so much the correctness of the design but rather the correctness of the implementation. Nevertheless, in reality, verification of the implementation is done much less rigorously. This makes it imperative that we focus more on developing techniques that enable us to verify actual code that will be compiled and executed. A major fraction of such code has been written, continues to be written and, in my opinion, will continue to be written in C.

Present day code validation falls in two broad categories - testing and formal verification. The merits and demerits of testing [33] are well-known and thus it is unnecessary to dwell on them in detail here. It suffices to mention that the necessity of being *certain* about the correctness of a piece of code precludes exclusive reliance on testing as the validation methodology, and forces us to adopt more formal approaches.

Software Verification

State of the art formal *software verification* is an extremely amorphous entity. Originally, most approaches in this field could be categorized as belonging to either of two schools of thought: theorem proving and model checking. In theorem proving (or deductive verification [27]), one typically attempts to construct a formula ϕ (in some suitable logic like higher-order predicate calculus) that represents both the system to be verified and the correctness property to be established. The validity of ϕ is then established using a theorem prover. As can be imagined, deductive verification is extremely powerful and can be used to verify virtually any system (including *infinite* state systems) and property. The flip-side is that it involves a lot of manual effort. Furthermore it yields practically no diagnostic feedback that can be used for debugging if ϕ is found to be invalid.

Model Checking

Where theorem proving fails, model checking [17] shines. In this approach, the system to be verified is represented by a *finite* state transition system $\mathcal{M}$ (often a Kripke structure) and the property to be established is expressed as a temporal logic [29] (usually CTL [12] with fairness or LTL [28]) formula ϕ. The model checking problem is then to decide whether $\mathcal{M}$ is a model of ϕ. Not only can this process be automated to a large degree, it also yields extremely useful diagnostic feedback (often in the form of counterexamples) if $\mathcal{M}$ is found not to model ϕ. Owing to these and other factors, the last couple of decades have witnessed the emergence of model checking as the eminent formal verification technique. Various kinds of temporal logics have been extensively studied [21] and efficient model checking algorithms have been designed [14, 34]. The development of techniques like *symbolic* model checking [7], *bounded* model checking [4, 5], *compositional* reasoning [13] and *abstraction* [15] have enabled us to verify systems with enormous state spaces [8].

One of the original motivations behind the development of model checking was to extract and verify *synchronization skeletons* of concurrent programs, a typical software verification challenge. Somewhat ironically, the meteoric rise of model checking to fame was largely propelled by its tremendous impact on the field of hardware verification. I believe that a major factor behind this phenomenon is that model checking can only be used if a finite model of the system is available. Also since real system descriptions are often quite large, the models must be extracted automatically or at least semi-automatically. While this process is often straightforward for hardware, it is much more involved for software. Typically software systems have *infinite* state spaces. Thus, extracting a finite model often involves a process of abstraction as well.

Predicate Abstraction

For a long time, the applicability of model checking to software was somewhat handicapped by the absence of powerful automated model extraction techniques. This scenario changed with the advent of predicate abstraction [22] (a related notion called data type abstraction used by systems like Bandera [2, 20] can be viewed as a special instance of predicate abstraction). Even though predicate abstraction was quickly picked up for research in hardware verification as well [18, 19], its effect on code verification was rather dramatic. It forms the backbone of two of the major code verifiers in existence, SLAM [1, 35] and BLAST [6, 24].

Predicates abstraction is parameterized by a set of predicates involving the variables of the concrete system description. It also involves non-trivial use of theorem provers (in fact the its original use [22] was to create abstract state transition graphs using the theorem prover PVS). Thus it has triggered a more subtle effect - it has caused the boundary between model checking and theorem proving to become less distinct.

Challenge 1 *Predicate abstraction essentially works by aggregating system states that are similar in terms of their data valuations. It is insensitive to the events that a system can perform from a given state. Can we develop other notions of abstraction that leverage the similarities between system states in terms of their dynamic (event-based) behavior? Such abstractions would complement predicate abstraction and lead to further reduction of state-space size.*

Abstraction Refinement

Even with progress in automated model extraction techniques, verifying large software systems remains an extremely tedious task. A major obstacle is created by the abstraction that happens during model extraction. Abstraction usually introduces additional behavior that is absent in the concrete system. Suppose that the model check fails and the model checker returns a counterexample CE. This does not automatically indicate a bug in the system because it is entirely possible that CE is an additional behavior introduced by abstraction (such a CE is often called a *spurious* counterexample). Thus we need to verify whether CE is spurious, and if so we need to refine our model so that it no longer allows CE as an admissible behavior. This process is called abstraction refinement. Since the extracted models and counterexamples generated are quite large, abstraction refinement must be automated (or at least semi-automated) to be practically effective.

The above requirements lead naturally to the paradigm called counterexample guided abstraction refinement (CEGAR). In this approach, the entire verification process is captured by a three step *abstract-verify-refine* loop. The actual details of each step depend on the kind of abstraction and refinement methods being used. The steps are described below in the context of predicate abstraction, where $Pred$ denotes the set of predicates being used for the abstraction.

1. **Step 1 : Model Creation.** Extract a finite model from the code using predicate abstraction with $Pred$ and go to step 2.
2. **Step 2 : Verification.** Check whether the model satisfies the desired property. If this is the case, the verification successfully terminates; otherwise, extract a counterexample CE and go to step 3.

3. **Step 3 : Refinement.** Check if CE is spurious. If not we have an actual bug and the verification terminates unsuccessfully. Otherwise we improve $Pred$ and go to step 1. Let us refer to the improved $Pred$ as $\overline{Pred}$. Then $\overline{Pred}$ should be such that CE and all previous spurious counterexamples will be eliminated if the model is extracted using $\overline{Pred}$.

Challenge 2 *Software model checking has focused almost exclusively on the verification of safety properties via some form of trace containment. It would be desirable to extend its applicability to more general notions of conformance such as simulation and richer class of specifications such as liveness.*

Challenge 3 *The complexity of predicate abstraction is exponential in the number of predicates used. The naive abstraction refinement approach keeps on adding new predicates on the basis of spurious counterexamples. Previously added predicates are not removed even if they have been rendered redundant by predicated discovered subsequently. Can we improve this situation?*

Compositional Reasoning

CEGAR coupled with predicate abstraction has become an extremely popular approach toward the automated verification of sequential software, especially C programs [6] such as device drivers [35]. However, considerably less research has been devoted to-wards the application of these techniques for verifying concurrent programs.

Compositional reasoning has long been recognized as one of the most potent solutions to the state-space explosion which plagues the analysis of concurrent systems. Compositionality appears explicitly in the theory of process algebras such as CSP [26], CCS [31] and the π-Calculus [32]. A wide variety of process algebraic formalisms have been developed with the intention of modeling concurrent systems and it is therefore natural [3] to investigate whether process algebraic concepts are useful in the verification domain as well.

One of the key concepts arising out of the process algebraic research is the need to focus on communication [31] when reasoning about concurrent systems. For instance CSP advocates the use of *shared actions* as the principal communication mechanism between concurrent components of a system. Moreover, shared action communication can model message-passing C programs such as client-server systems and web-services in a very natural manner.

Challenge 4 *The CEGAR paradigm has been used with considerable success on sequential programs. Can we also use it to compositionally verify concurrent programs? What, if any, are the restrictions that we might need to impose in order to achieve this goal?*

State/event based Analysis

A major difficulty in applying model checking for practical software verification lies in the modeling and specification of meaningful properties. The most common instantiations of model checking to date have focused on finite-state models and either branching-time (CTL [12]) or linear-time (LTL [28]) temporal logics. To apply model checking to software, it is necessary to specify (often complex) properties on the finite-state abstracted models of computer programs. The difficulties in doing so are even more pronounced when reasoning about *modular* software, such as concurrent or component-based sequential programs. Indeed, in modular programs, communication among modules proceeds via actions (or events), which can represent function calls, requests and acknowledgments, etc. Moreover, such communication is commonly data-dependent. Software behavioral claims, therefore, are often specifications defined over combinations of program actions and data valuations.

Existing modeling techniques usually represent finite-state machines as finite annotated directed graphs, using either *state-based* or *event-based* formalisms. It is well-known that the two frameworks are interchangeable. For instance, an action can be encoded as a change in state variables, and likewise one can equip a state with different actions to reflect different values of its internal variables. However, converting from one representation to the other often leads to a significant enlargement of the state space. Moreover, neither approach on its own is practical when it comes to modular software, in which actions are often data-dependent: considerable domain expertise is then required to annotate the program and to specify proper claims.

Challenge 5 *Can we develop a formalism for succinctly expressing and efficiently verifying state/event-based properties of programs? In particular we should be able to verify a state/event system directly without having to translate it to an equivalent pure-state or pure-event version. Further, can we combine state/event-based analysis with a compositional CEGAR scheme?*

Deadlock Detection

Ensuring that standard software components are assembled in a way that guarantees the delivery of reliable services is an important task for system designers. Certifying the absence of deadlock in a composite system is an example of a stringent requirement that has to be satisfied before the system can be deployed in real life. This is especially true for safety-critical systems, such as embedded systems or plant controllers, that are expected to always service requests within a fixed time limit or be responsive to external stimuli.

In addition, many formal analysis techniques, such as temporal logic model checking [12, 17], assume that the systems being analyzed are deadlock-free. In order for the results of such analysis to be valid, one usually needs to establish deadlock freedom separately. Last but not least, in case a deadlock is detected, it is highly desirable to be able to provide system designers and implementers with appropriate diagnostic feedback.

However, despite significant efforts, validating the absence of deadlock in systems of realistic complexity remains a major challenge. The problem is especially acute in the context of concurrent programs that communicate via mechanisms with blocking semantics, e.g., synchronous message-passing and semaphores. The primary obstacle is the well-known *state space explosion* problem whereby the size of the state space of a concurrent system increases exponentially with the number of components.

As mentioned before, two paradigms are usually recognized as being the most effective against the state space explosion problem: *abstraction* and *compositional reasoning*. Even though these two approaches have been widely studied in the context of formal verification [15, 23, 25, 30], they find much less use in deadlock detection. This is possibly a consequence of the fact that deadlock is inherently non-compositional and its absence is not preserved by standard abstractions. Therefore, a compositional CEGAR scheme for deadlock detection would be especially significant.

Challenge 6 *In the light of the above discussion, can we develop a compositional*

CEGAR-based procedure for deadlock detection?

Summary

This dissertation presents a framework for verifying concurrent message-passing C programs with specific emphasis on addressing the challenges enumerated earlier in this article. Among other things, we addresses Challenge 5 by enabling both state-based and action-based properties to be expressed, combined, and efficiently verified. To this end we propose the use of *labeled Kripke structures* (LKSs) as the modeling formalism. In essence, an LKS is a finite state machines in which states are labeled with atomic propositions and transitions are labeled with events (or actions). In the rest of this article we will refer to a concurrent message-passing C program as simply a program.

Our state/event-based modeling methodology is described in two stages. We first present a semantics of programs in terms of LKSs. We then develop a generalized form of predicate abstraction to construct conservative LKS abstractions from programs in an automated manner. We provide formal justification for our claim that the extracted LKS models are indeed conservative abstractions of the concrete programs from which they have been constructed.

Subsequently we address Challenge 2 and Challenge 4 by presenting a compositional CEGAR procedure for verifying simulation conformance between a program and an LKS specification. We define the notion of witness LKSs as counterexamples to simulation conformance and present algorithms for efficiently constructing such counterexamples upon the failure of a simulation check. We next present algorithms for checking the validity of witness LKSs and refining the

LKS models if the witness is found to be spurious. The entire CEGAR procedure is compositional in the sense that the model construction, witness validation and abstraction refinement are performed component-wise.

Moving on, we propose the use of predicate minimization as a solution to Challenge 3. Our approach uses pseudo-Boolean constraints to minimize the number of predicates used for predicate abstraction and thus eliminates redundant predicates as new ones are discovered. We also present an action-guided abstraction refinement scheme to address Challenge 1. This abstraction works by aggregating states based on the events they can perform and complements predicate abstraction naturally. Both these solutions are seamlessly integrated with the compositional CEGAR scheme presented earlier.

We also present the logic SE-LTL, a *state/event* derivative of the standard linear temporal logic LTL. We present efficient SE-LTL model checking algorithms to help reason about state/event-based systems. We also present a compositional CEGAR procedure [9, 10, 16] for the automated verification of concurrent C programs against SE-LTL specifications. SE-LTL enriches our specification mechanism by allowing state/event-based liveness properties and is thus relevant to both Challenge 2 and Challenge 5.

Finally, we address Challenge 6 by presenting a compositional CEGAR scheme to perform deadlock detection on concurrent message-passing programs [11]. In summary, the demand for better formal techniques to verify concurrent and distributed C programs is currently overwhelming. This dissertation identifies some notable stumbling blocks in this endeavor and provides a road map to their solution.

Bibliography

[1] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, May 2001.

[2] Bandera website. `http://www.cis.ksu.edu/santos/bandera`.

[3] BEHAVE! website. `http://research.microsoft.com/behave`.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Y. Zue. *Bounded Model Checking*, volume 58 of *Advances in computers*. Academic Press, 2003. To appear.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, March 1999.

[6] BLAST website. `http://www-cad.eecs.berkeley.edu/~rupak/blast`.

[7] J.R. Burch, Edmund M. Clarke, David E. Long, K.L. MacMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[8] J.R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS '90)*, pages 1–33. IEEE Computer Society Press, 1990.

[9] S. Chaki, J. Ouaknine, K. Yorav, and Edmund M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proceedings of the 2nd Workshop on Software Model Checking*

(SoftMC '03), volume 89(3) of *Electonic Notes in Theoretical Computer Science*, July 2003.

[10] Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design (FMSD)*, 25(2–3):129–166, September – November 2004.

[11] Sagar Chaki, Edmund Clarke, Joël Ouaknine, and Natasha Sharygina. Automated, compositional and iterative deadlock detection. In *Proceedings of the 2nd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '04)*, pages 201–210. OMNI Press, June 2004.

[12] Edmund Clarke and Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, May 1981.

[13] Edmund Clarke, David Long, and Kenneth McMillan. Compositional model checking. In *Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS '89)*, pages 353–362. IEEE Computer Society Press, June 1989.

[14] Edmund M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and System (TOPLAS)*, 8(2):244–263, April 1986.

[15] Edmund M. Clarke, O. Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, September 1994.

[16] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, July 2000.

[17] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, Cambridge, MA, 2000.

[18] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In Mark Aagaard and John W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD '02)*, volume 2517 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, November 2002.

[19] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171. Springer-Verlag, July 1999.

[20] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Hongjun Zheng, and Willem Visser. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pages 177–187. IEEE Computer Society Press, May 2001.

[21] E. Allen Emerson. *Temporal and modal logic*, volume B: formal models and semantics of *Handbook of Theoretical Computer Science*, pages 995–1072. MIT Press, 1990.

[22] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997.

[23] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(3):843–871, May 1994.

[24] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '02)*, volume 37(1) of *SIGPLAN Notices*, pages 58–70. ACM Press, January 2002.

[25] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the 2000 International Conference on Computer-Aided Design (ICCAD '00)*, pages 245–252. IEEE Computer Society Press, November 2000.

[26] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.

[27] C.A.R Hoare. An axiomatic basis for computer programming. *Communications of the ACM (CACM)*, 12(10):576–580, October 1969.

[28] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '85)*, pages 97–107. ACM Press, January 1985.

[29] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

[30] Kenneth L. McMillan. A compositional rule for hardware design refinement. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35. Springer-Verlag, June 1997.

[31] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989.

[32] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[33] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[34] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani and U. Montanari, editors, *Proceedings of the 5th Colloquium of the International Symposium on Programming (ISP '82)*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, April 1982.

[35] SLAM website. `http://research.microsoft.com/slam`.