Model Check What You Can, Runtime Verify the Rest

Timothy L. Hinrichs, A. Prasad Sistla and Lenore D. Zuck

University of Illinois at Chicago
{hinrichs,sistla,zuck}uic.edu

Abstract

Model checking and runtime verification are pillars of formal verification but for the most part are used independently. In this position paper we argue that the formal verification community would be well-served by developing theory, algorithms, implementations, and applications that combine model checking and runtime verification into a single, seamless technology. This technology would allow system developers to carefully choose the appropriate balance between offline verification of expressive properties (model checking) and online verification of important parts of the system's state space (runtime verification). We present several realistic examples where such technology appears necessary and a preliminary formalization of the idea.

1 Introduction

The virtues and drawbacks of model checking (MC) are well known: model checked systems undeniably satisfy some important properties, but some systems are too unwieldy for model checking in practice. Likewise the virtues and drawbacks of runtime verification (RV) are well known: most systems can be runtime verified, but that verification only ensures that some important properties hold for that fraction of the system that happens to be exercised during verification. In short, model checking guarantees completeness at the cost of applicability and runtime verification guarantees applicability at the cost of completeness. But if there were techniques for integrating model checking and runtime verification (MC+RV), system developers would have fine-grained control over the tradeoff between completeness and applicability during the verification process.

Besides the intellectual gratification of hybridizing two well-known techniques for verification, MC+RV would allow the verification community to present a unified front to the outside world—to provide consumers of verification technology with a single conceptual product to understand and apply. Improving the accessibility of verification technology to the outside world will help both society as a whole, since software will contain fewer bugs, and the research community in particular because of the increased source of real-world verification problems encountered in the wild.

The technical problems arising when combining model checking and runtime verification are mainly due to their substantially different underlying assumptions. Model checking runs offline (*i.e.*, before the system is deployed), and runtime verification runs online (*i.e.*, after the system is deployed). Model checking only works properly when the entire system can be examined, and runtime verification only requires a single run of the system. These differences are of course what makes the hybridization of the two techniques so attractive.

These basic observations about the potential benefits of MC+RV are not novel in and of themselves. [7] employed static program analysis (of which MC is a special sort) to simplify runtime verification checks. [20] utilized the combination of model checking and runtime verification to identify security vulnerabilities in web applications. [8] utilizes runtime analysis to

perform an anytime variant of static analysis. While such examples of combining static and dynamic analysis exist, they are not based on a single, unified theory of combining these forms of analysis and require intimate knowledge of the underlying analysis (or verification) technology.

In this position paper, we advocate investigating the underlying theory of combining model checking and runtime verification. That theory will enable tools where a developer chooses the extent of verification appropriate to the application—to prioritize how mission critical each portion of a system is and to verify that system accordingly. In short, we argue that verification technology will become commonplace only when that technology accounts for the limited resources (time, compute cycles, *etc.*) organizations have to devote to verification.

In the remainder of this paper, we begin by comparing the strengths and weaknesses of model checking and runtime verification in more detail (Section 2). We then state the position this paper puts forth (Section 3) and identify several domains where the combination of model checking and runtime verification appears to offer substantial benefit (Section 4). Then we briefly discuss a formalization of the central problem of MC+RV (Section 5) and finish with related work (Section 6) and a conclusion (Section 7).

2 Model Checking and Runtime Verification

Model Checking. Model checking is a technique where one takes a model of a system and a model of a property and algorithmically checks whether the system satisfies the property [10, 11, 27]. Typically, the systems one has in mind are hardware or software systems, and the specifications are temporal. While model checking is commonly applied to finite-state systems, at times it is possible to apply it for infinite-state systems using, for example, abstractions, symmetry properties, and small-model properties (e.g., [1, 14, 21, 25].) Model checking can be *explicit-state* where both model and properties (or their complements) are represented as automata-like structures, and model checking reduces to finding whether any path in the system violates the property. Of course, when liveness properties are concerned, one must consider only fair paths. An alternative approach to model checking is using *symbolic* techniques (most often, BDD- or SMT- based). Each approach has its strengths. For example, all systems we are aware of that model checked timed properties employ an explicit-state approach.

The attraction of model checking is that it is algorithmic, the alternative being the use of deductive methods and theorem provers (assuming that one wishes the verification to be at least automatically checked). However, whichever model checking approach one employs, state-explosion remains the main hurdle of using model checking for large systems. Much work has been devoted to pushing the envelope, to the extent that some infinite state systems have been model checked. Yet, it is still often the case, especially with software, that the state space is too large and no known reduction in the state space removes this problem. This is the reason that model checking is mainly used as a debugging mechanism in the early design stages and methods such as bounded model checking [6] have gained popularity.

Another drawback of model checking techniques is that they are not easily amenable to compositionally. While much work as been done on compositional model checking [21], the application of such methods in "full-fledged" software systems is still impractical. This is important since a single system often includes several components that can be swapped out for similarly behaving components, and one may wish to verify the system without the swappable components independently from the components themselves. To complicate matters further, some of the swappable components may be *black boxes*. Lack of model checking compositionality is also problematic for large (*e.g.*, infinite state) systems where there is a good reason to believe that during most of its executions the system visits only a small number of states, and this

portion of the system can be model-checked.

For these and other reasons (some of them will be elaborated on in the sequel) one may want to use model checking to verify only parts of a system and to use other methodologies to guarantee the correctness of other parts. We propose to use Runtime Verification for this purpose.

Runtime Verification. In contrast to model checking, runtime verification monitors a system and extracts information from its executions to detect (and, sometimes, react to) flaws. While the term *runtime verification* is only a decade old, the ideas behind runtime verification had been used long before. Recently, runtime verification has become an active research area because of its power to ensure correctness of systems that are not amenable to model checking. In particular, while model checking is mostly useful at the design stage, runtime verification is useful at the deployment stage [3, 4, 12, 19].

The main technical problems in runtime verification are (i) automatically adding runtime verification code to an existing system and (ii) minimizing the overheads of that code. A popular approach for the former is to use aspect-oriented programming techniques that allow such code to be written independently of the original system and through compiler techniques automatically injected into the code [2]. For the second problem, some work has employed static analysis to eliminate any runtime checks that can easily be identified as unnecessary [7].

In contrast to model checking, runtime verification inherently performs well when exploring only a portion of the state space—it was designed to do so. But its overall guarantees are much weaker than those offered by model checking. The properties that are checked need only hold over the portion of the state space that are executed once the system is deployed, and the class of properties amenable to runtime verification is a strict subset of the properties amenable to model checking. Thus while runtime verification is better behaved on partial state space explorations, it offers weaker guarantees of completeness (both in terms of expressiveness and state-space coverage).

3 Position

Both runtime verification and model checking are powerful techniques for checking correctness of systems. Table 1 represents a rough comparison of the methods.

	Model Checking	**Runtime Verification**
Completeness	Entire State Space	Pertinent State Space
Applicable Properties	Arbitrary (temporal)	Safety+
Performance Penalty	Offline	Online

Table 1: Model Checking vs. Runtime Verification

Our position is that a theory for combining model checking and runtime verification offers numerous advantages and advances the community's long term goal of making both hardware and software system verification ubiquitous. Our premise is that it is often relatively easy to identify a set of states where model checking techniques can be employed. This set of space can then be model checked. Once a system enters a state outside of this set, a runtime monitor can be invoked. Of course, one has to monitor whether or not the system's state is in or out of this set. Thus, the test whether or not the system is in a state that is in the set has to be simple, or otherwise, the advantage of model checking states in the set is compromised. In the

sequel we give examples of several systems/properties where MC+RV is beneficial. Below we describe MC+RV in terms of the three evaluation criteria from Table 1.

Completeness: Model checking is concerned with the entire state space of the system, whereas runtime verification is only concerned with the portion of the state space that is relevant to a particular deployment. By combining model checking and runtime verification the state space explored is somewhere in between the entire space and the deployment-relevant space.

Applicable Properties: Model checking allows us to verify temporal properties of a system whereas runtime verification is usually limited to safety properties. A hybrid of the approaches would allow us to verify all that runtime verification can outside of the model checked state space, and all that model checking can in it.

Performance Penalty: The performance penalty for model checking occurs offline (*i.e.*, before deployment), whereas runtime verification pays an online performance penalty. The performance penalty of the hybrid approach is a developer-controlled combination of offline and online.

We envision the results of this theory embodied within a tool that allows a developer to choose the extent to which model checking is performed, allowing runtime verification to handle the remaining system. Our message to developers is simple: model check what you can, and runtime verify the rest.

4 Examples

Here we present a number of examples illustrating why combining model checking and runtime verification (MC+RV) is useful for verifying real-world systems. For each example, we ask the questions:

- What portion of the space should we model check and which portion should we runtime verify?
- What formula are we model checking, and what formula are we runtime verifying?

4.1 Contracts

In the context of programming languages, a software contract is a logical specification dictating how a piece of software is supposed to behave [22]. For example, a Heap is a data structure that stores an ordered set of elements. The contract for the Heap might require that inserting an element produces a heap that represents the same set as before the insert but with the new element added; merging two heaps results in a heap with the union of the original heaps' elements.

In theoretical computer science, researchers will often introduce a new algorithm and then prove its correctness. Those proofs of correctness usually assume the data structures they utilize behave as they are supposed to, *e.g.*, the Heap obeys its contract. The benefit to this style of proof is that any implementation of those data structures that satisfies their contracts can be used without violating the correctness of the overall algorithm. For example, [15] describes an algorithm for curve finding useful for computer vision that utilizes a Heap and proves correctness as well as runtime complexity assuming the Heap satisfies its contract and complexity bounds.

Partitioning the proof of an algorithm's correctness into two phases (algorithm and data structures) points to an application of MC+RV. For systems relying on data structures, we can model check the correctness of those systems assuming the data structures behave according to their contracts. We can then independently verify the contracts of the data structures. For those situations where offline contract verification is impractical, we can runtime verify the contracts, which is functionality being built into modern programming languages. [16] describes an implementation of runtime contract verification that for certain properties preserves the (amortized) asymptotic behavior of the data structures.

In this example, it is clear what portion of the system ought to be model checked and what portion should be runtime verified. It is also clear what formula we intend to model check and what formula we should runtime verify: if the data structure's contract is α and the overall property of the system we want to verify is β then we model check $\alpha \rightarrow \beta$ and we runtime verify α. At runtime, the verifier is turned on only during the data structure operations.

4.2 Race Conditions

Race conditions occur in multi-threaded software when the integrity of data depends on the order in which different threads access that data. For example, if Alice and Bob both try to withdraw the last $100 from their joint bank account, and there is some way for them both to succeed, then there is a race condition in the bank's software.

The authors of [13] outfitted Java with a runtime race condition detector and found that the overheads could be an order of magnitude or more slowdown of the software. This performance penalty is likely too high to warrant automatic race condition detection in production systems, especially when the race conditions are rare. However, if we could verify that race conditions do not occur "most of the time" via model checking, we might be willing to pay such overheads for runtime race condition detection in those rare situations when they might occur.

For example, suppose that the bank software correctly handled the situation where Alice and Bob were both simultaneously trying to withdraw their last $100 but not the case where Alice, Bob, and their son Charlie were all simultaneously trying to withdraw the last of their money. If we could verify offline that the software performed correctly in the case of two competitors, we might pay for an order of magnitude slowdown in that exceedingly rare situation when three or more competitors accessed the system simultaneously.

Since the state space of the system grows exponentially with the numbers of competitors, one would wish to model check on as few competitors as possible. Yet, if there are good reasons to believe that the number of simultaneous competitors is usually small, one can employ MC+RV by model checking for this small number of competitors and runtime verify when the number of competitors is higher.

To apply MC+RV the developer must decide what to model check and what to runtime verify since she is the only one who knows the number of competitors that is rare enough not to warrant model checking. The formula that we must model check is the obvious one: if β represents a lack of race conditions and α represents that the number of competitors is not exceedingly rare then we want to model check $\alpha \rightarrow \beta$, and we runtime verify α.

To investigate this idea, we designed a variant of Peterson 2-process mutual exclusion algorithm [24] that prevents two processes from accessing a *critical section* simultaneously. In our version, there are N threads (for any $N > 1$), and the goal is to guarantee mutual exclusion when the number of competitors for access to the critical section is ≤ 2. The protocol is presented in Figure 1.

There, N processes run asynchronously. There is a global variable *last* that can take on

$$
\begin{array}{lll}
\textbf{in} & N & : \ \textbf{natural where} N > 1 \\
\textbf{local} & last & : \ [1..N] \\
\Big|\Big|_{id=1}^{N} P[id] :: & \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} 0 & \textbf{idle} \\ 1 & last := id \\ 2 & \textbf{wait until } \forall j \neq id.(loc[j] = 0 \vee turn \neq id) \\ 3 & \textbf{critical section} \end{array}\right] \end{array}\right] &
\end{array}
$$

Figure 1: A Variant of Peterson's 2-process Mutual Exclusion Protocol

values between 1 and N and denotes the id of the last process to wish to enter the critical section. Each process can be in one of four locations: location 0 (initial state) where it may idle forever or wish to enter the critical section, location 1 where it declares that it wishes to enter the critical section by setting *last* to its own id, location 2 where it waits until no other process is competing or is in the critical section (*i.e.*, all are in location 0) *or* that *last* has changed since the process last modified it, hence, it is not the last one to enter the competition. In this case, the process can enter the critical section (location 3), after which it returns to its idle state.

We formally proved, using the method of invisible invariants [25] with the BDD-based model checker TLV that

$$\Box(\sum_{i=1}^{N} at\_\ell_{[1..3]}[i] \leq 2 \ \longrightarrow \ \forall i, j.\ (i \neq j \rightarrow \neg(at\_\ell_3[i] \ \wedge \ at\_\ell_3[j])))$$

where $at\_\ell_k[i]$ denotes that the process whose id is i is at location k. The summation term counts the processes whose location is in the range 1..3, *i.e.*, the number of competitors.

Thus, as long as the number of competing threads is no greater than 2, mutual exclusion is guaranteed (via model checking). To identify all violations of mutual exclusion, we need only outfit the system with a runtime verifier that performs mutual exclusion detection when the number of threads is greater than 2[1].

Following the terminology of the previous subsections, here α is the property that the number of competitors is less than 2, and β is the mutual exclusion property. We model check $\alpha \rightarrow \beta$ and we runtime verify α.

4.3 Hardware Verification

[26] describes IDV, a system developed at Intel that enables simultaneous hardware logic design and verification. One of the keys to efficiency in this work is giving the developer the ability to provide the model checker with hints about the behavior of certain aspects of the design. For example, the developer can tell the model checker that two intermediate results are never simultaneously 1. This additional information helps prune the state space that must be verified; however, if the developer makes a mistake when providing the hints, the property that has been verified by the model checker may not hold on the actual system.

Developer-provided hints implicitly partition the state space of the system. Some of the state space can be used to model check high level properties of the design, and other portions

[1]It is interesting to note that the protocol of Figure 1 satisfies liveness, i.e., that $\forall i.\ \Box(at\_\ell_1[i] \ \longrightarrow \ \Diamond at\_\ell_3[i])$

of the state space are used to verify lower level properties. For those situations where the hints are beliefs about how the environment will interact with the system, it is natural to use runtime verification to verify the lower level properties—to analyze whether or not the developer-provided hints are satisfied by the environment. Sufficient testing will uncover any flaws in the developer-provided hints.

In this example, the partitioning of the space into that which is model checked and that which is runtime verified is a direct result of the hints the developer provides. If the developer asserts hints $\alpha_1,\ldots,\alpha_n$, and we want to model check the property β then the model checking query we actually analyze is $\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta$. The properties we verify at runtime are just $\alpha_1, \ldots, \alpha_n$.

4.4 Web Services

Many web applications today rely on external web services. For example, an e-commerce web site will validate a credit card number by consulting the appropriate company's web service. Verifying that an e-commerce web site behaves appropriately is difficult because the code for external web services is unknown (and may change from the time the application is verified to the time it is deployed). Nevertheless, it is reasonable for a web developer to hope that verification technology can help give her confidence that her code behaves appropriately.

Here MC+RV is useful because model checking helps the developer verify that the web application is well-behaved assuming that the credit card company's web service is well-behaved. For example, if the web service is supposed to return a response in 15 milliseconds, and the web application is a real-time system that relies on that fact, then it can be model checked to ensure its real-time properties hold so long as the credit card company's service behaves as advertised. By runtime monitoring the credit card company's web service, we can detect when it takes longer than 15 milliseconds and identify situations in which the overall application may miss its real-time constraints through no fault of its own.

In this example the partitioning of the system's state space is based on the code available at the time of model-checking. The code that exists can be used for model checking; the remainder must be runtime verified. This kind of model checking requires additional information from the developer: a spec that dictates how the external web service is intended to behave. The model checking property of interest is again $\alpha \rightarrow \beta$, where α is the spec for the web service and β is the high-level behavior of the web application. At runtime we simply verify α holds over each web service execution.

4.5 Repurposed Component

Many software applications delegate some low-level functionality to the operating system in which they are deployed. In Linux, for example, it is common to read the contents of a directory by executing the `ls` command. If the application interacts with the operating system by simply executing a system command written as a string, it may seem as though any possible OS command is being used. This is problematic from the perspective of model checking because now the entire state space of the application plus the entire state space of the operating system would need to be checked.

Such problems can sometimes be addressed with MC+RV. If through simple analysis or by developer intervention we learn that the application only uses one or two OS commands, we can model check the application and the implementations for those commands, without needing to model check the entire state space of the OS. We can then runtime verify that the application only invokes the OS commands that we model checked.

In this example, we model check $\alpha \rightarrow \beta$ where α says the only OS commands are those on the developer's list, and β is the application property of interest. At runtime we monitor α: that the only OS commands executed are those from the list.

4.6 Exceptions

Computer systems can conceptually be broken into two kinds of code: exception-handling code and everything else. We often expect exception-handling code to be run infrequently; we also expect that many properties that we want to verify do not depend on exception-handling code. It is therefore reasonable to model check the system assuming no exceptions are triggered and to relegate the verification of the exception-handling code to runtime.

In this example, it is clear how to partition the state space into the parts that we model check and runtime verify. For model-checking, consider all paths through the code that produce no exceptions, and for run-time verification, place runtime verifiers only inside exception handling code. It is also clear that if we want property β to hold that we model check β. What is less clear is how we runtime verify β in the exception-handling code. If β is a liveness property, run-time verification will not be possible. But even if β is a safety property, we may need to outfit the application to store the pertinent history so that the exception-handling code has access to the right information when performing runtime verification.

5 Discussion

From a formal perspective, we can view MC+RV as an instantiation of assume-guarantee reasoning. Given a system $\mathcal{S}$ and a property of interest β, an instance of MC+RV constructs a (safety) formula α, model checks that $\mathcal{S}$ guarantees β assuming α and then runtime verifies α. We know that any execution satisfying α is guaranteed to satisfy β because of the model checker, but we do not know whether the remaining executions satisfy or falsify β. The runtime verifier identifies executions for which α is falsified and hence additional action must be taken, *e.g.*, issue a warning, runtime verify β, or even verify some other property γ related to β. Note that runtime verifying β (or γ) may require having stored some history information in the state even before α is falsified.

A key observation about this formulation is that α must be a safety property, since it needs to be runtime verified; furthermore, runtime verifying α must be less expensive than runtime verifying β itself since otherwise we could forego model checking entirely and simply verify β. In fact, α should be a formula that is the least expensive to verify out of all the candidate αs. Each of the examples discussed earlier fits within this conceptual framework, and each example being distinguished in terms of what constitutes α, what constitutes β, and what the application does if the runtime verifier discovers that α is falsified.

For software contracts (and web services), α dictates that the implementations of data structures (*e.g.*, Heap) used within the application (*e.g.*, curve-finding) are correct, and β is just the property of interest. Thus for this example we model check that if the data structure implementation is correct (α is satisfied) then the application is correct (β is satisfied), and we runtime verify that the data structure implementation is correct. If the runtime verifier discovers that α is falsified, we have found a bug in the data structure implementation, which is irrelevant for the purpose of curve-finding correctness.

For the race conditions example, α is the condition that the number of threads throughout the execution is 2 or fewer, and β says there are no race conditions. Thus the model checker ensures that when the number of threads is 2 or fewer (α), no race conditions hold (β), and

we runtime verify that there are 2 or fewer threads. When the runtime verifier discovers α is falsified, in this example we runtime verify β: that no race conditions occur.

For hardware verification, α is the conjunction of the developer-provided hints dictating invariants on parts of the system. β is the property of interest. We model check that when the developer hints hold, the property of interest also holds, and we runtime verify that the hints are correct. If the runtime verifier discovers that α is falsified, we simply inform the developer that some of her hints were wrong, thereby suggesting that the hints be altered and the model checking rerun.

For the repurposed component, α dictates that only a handful of system calls are ever made by the program. β is the property of interest. We therefore model check the system assuming that the only system calls are the ones we have identified, and we runtime verify that those are the only system calls the application makes. When α is violated at runtime, we know that β may (or may not) be violated, and the appropriate action depends on the application.

For exceptions, α dictates that no exceptions are thrown (*i.e.*, it is the negation of the exception conditions), and β is the property of interest. We therefore model check the system assuming that no exceptions are thrown, and we runtime verify that no exceptions are thrown. When α is violated (exceptions are thrown), a potentially different formula γ is likely checked, *e.g.*, all the appropriate files are closed and the program terminates.

In the future, we plan to investigate automated techniques for generating α automatically, as well as optimizing the choice of α so as to minimize the cost of its runtime monitor while simultaneously minimizing the cost of model checking $\alpha \rightarrow \beta$. We can then utilize existing runtime verification techniques to automatically instrument the system $\mathcal{S}$ to monitor α and existing techniques to model check $\alpha \rightarrow \beta$.

6 Related Work

Below we detail work that combines static and dynamic analysis of systems.

One line of work attempting to understand when and how we can verify properties of an entire state space while only exploring a portion of that space is the work on 3-valued model checking [17]. The key insight to this work is that if we know what we don't know then sometimes we can infer what the possible things we don't know could be and conclude that none of those possibilities could violate the specification. Of course, sometimes we simply fail to explore enough of the space for even 3-valued model checking to help us verify the system.

One class of related work utilizes static analysis to simplify runtime verification code *e.g.*, [7]. After statically analyzing code to extract invariants, those invariants can demonstrate that particular runtime verification checks are unnecessary because the code enforces them directly.

Similarly [23] describes CCured, a C program transformation tool that attempts to impose a statically-checked type system on C programs. Those elements of a program that cannot be statically type-checked are outfitted with runtime monitors to catch memory errors. This work could be construed as using static techniques to simplify runtime checks as in [7]; however, the focus here is just the reverse: that the runtime checks are used only when the static analysis is insufficient.

Another line of work, *e.g.*, [5, 8, 9, 18, 20, 28], combines model generation, model checking, and testing (which can be construed as a kind of runtime verification) to achieve better coverage of the system's state space and find bugs or provide proofs of verification.

7 Conclusion

We propose investigating the theory of integrating model checking and runtime verification, with the eventual goal of providing a single framework for exporting verification technology to the outside world. From the perspective of model checking, MC+RV broadens the applicability of model checking to include systems that are too large or too dynamic to be completely explored. The downside is that the entire system cannot be model checked before it is deployed, and hence verification failures may not be found until after deployment. From the perspective of runtime verification, MC+RV expands the class of properties that can be verified and reduces the overhead of runtime verification. The downside is that the MC step may find bugs that are irrelevant to the deployments of interest and incurs offline costs. MC+RV therefore has drawbacks as well as benefits; hence, we envision the theory of MC+RV resulting in controls for the developer to choose the appropriate mix of model checking and runtime verification. For those situations where offline costs must be small, the developer can dictate the model checking component only consider a small portion of the state space, and for those situations where verification before deployment is important, the developer can dictate that only those components of the system not known at the time of analysis should be runtime verified. MC+RV will allow developers to choose the verification approach best-suited for their application by properly balancing the tradeoffs of model checking and runtime verification.

References

[1] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. In *N. Halbwachs and D. Peled, editors,* Proc. 11^{th} Intl. Conference on Computer Aided Verification (CAV'99)*, volume 1633 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 134–145, 1999.

[2] `http://www.eclipse.org/aspectj/`.

[3] Howard Barringer, Dov M. Gabbay, and David E. Rydeheard. From runtime verification to evolvable systems. In *RV*, pages 97–110, 2007.

[4] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From Eagleto RuleR. In *RV*, pages 111–125, 2007.

[5] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 3–14, 2008.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proc.* 7^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), volume 1579 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 193–207, 1999.

[7] Eric Bodden, Patrick Lam, and Laurie J. Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2010.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS*, 2006.

[9] Chia Y. Cho, Domagoj Babic, Pongsin Poosankam, Kevin Z. Chen, Edward X. Wu, and Dawn Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the Usenix Security Symposium*, 2011.

[10] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer-Verlag, 1981.

[11] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 8:244–263, 1986.

[12] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[13] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race-aware Java runtime. *Commun. ACM*, 53(11):85–92, 2010.

[14] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model checking under fairness assumptions. *ACM Trans. Prog. Lang. Sys.*, 19(4), 1997. Preliminary version appeared in 7th CAV, 1995.

[15] Pedro Felzenszwalb and David McAllester. A min-cover approach for finding salient curves. In *IEEE Workshop on Perceptual Organization in Computer Vision*, 2006.

[16] Robert Bruce Findler, Shu yu Guo, and Anne Rogers. Lazy contract checking for immutable data structures. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 111–128. Springer, 2007.

[17] Patrice Godefroid and Radha Jagadeesan. Automatic abstraction using generalized model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 137–150, London, UK, UK, 2002. Springer-Verlag.

[18] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: A new algorithm for property checking. In *Proceedings of the ACM Transactions on Software Engineering and Methodology,*, 2006.

[19] Klaus Havelund. Runtime verification of C programs. In *TestCom/FATES*, pages 7–22, 2008.

[20] Monica S. Lam and Michael Martin. Securing web applications with static and dynamic information flow tracking. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 3–12, 2008.

[21] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999.

[22] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2000.

[23] George C. Necula, Jeremy Condit, Matthew Harren, Scott Mcpeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27:2005, 2005.

[24] G. L. Peterson. A new solution to Lamport's concurrent programming problem. *ACM Trans. Prog. Lang. Sys.*, 5(1):56–65, 1983.

[25] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. 7^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of* Lect. Notes in Comp. Sci*., Springer-Verlag*, pages 82–97, 2001.

[26] Carl Seger. Treating constraints as components: An experiment in user control. Invited talk at the Seventh International Workshop on Constraints in Formal Verification, 2011.

[27] J. Sifakis and J.P. Queille. Fairness and related properties in transition systems – a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

[28] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: Better together. In *Software Testing and Analysis (ISSTA*, pages 145–156. ACM, 2006.