The Localization Reduction and Counterexample-Guided Abstraction Refinement

Edmund M. Clarke[1], Robert P. Kurshan[2], and Helmut Veith[3]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
[2] Cadence Design Systems, Inc., New York, NY 10014
[3] Formal Methods in Systems Engineering, Vienna University of Technology, Austria

Abstract. Automated abstraction is widely recognized as a key method for computer-aided verification of hardware and software. In this paper, we describe the evolution of counterexample-guided refinement and other iterative abstraction refinement techniques.

1 Introduction

The state explosion problem is still the major disadvantage of Model Checking. One of the most successful ways of dealing with this problem is to use abstraction to create a conservative abstraction and check it instead of the original model. Finding the right abstraction is nontrivial. If the abstraction is too coarse, there may be false negatives. On the other hand, if the abstraction is too precise, the resulting model may still be too large. One often used approach for finding a satisfactory model is to combine abstraction and refinement in an iterative manner. The first such iterative procedure was the Localization Reduction for models with state variables proposed by R. Kurshan in his 1995 book Computer-Aided Verification of Coordinating Processes [Kur94]. At CAV 2000, E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith generalized the Localization Reduction in a paper entitled Counterexample-Guided Abstraction Refinement [CGJ$^+$00]. Their new method, called CEGAR, combined the iterative refinement principle of the localization reduction with ideas from abstraction-based model checking [CGL94] and predicate abstraction [GS97] into a model checking framework for arbitrary Kripke structures and universal CTL* formulas that is also applicable to infinite-state systems, and thus, to software.

In this paper, we describe the evolution of counterexample-guided refinement and other iterative abstraction refinement techniques with an emphasis on the early history of the method. We also discuss later developments such as CEGAR for software and alternative approaches for generating good abstractions that do not involve the generation of counterexamples.

Sections 2 and 4 are contributed by Bob Kurshan, Section 3 by Ed Clarke and Helmut Veith.

2 Localization Reduction

Localization reduction [Kur94, Kur00] evolved from an algorithm for the verification of timed automata, based on successive approximations [AIKY93] (inspired by Newton's method). It provides an iterative algorithm for design abstraction, relative to a property to be verified.

The localization reduction algorithm iterates over abstractions determined by counterexamples on successive refinements. Ed Clarke *et al* used the iterative refinement technique in CEGAR [CGJ$^+$00] described in the following section. There are many interesting related procedures, *e.g.*, [DD01, JM05]. In a significant SAT-based improvement [MA03], the successive abstractions are determined not by the counterexamples but by the unSAT core used to refute falsification in the original model at a given depth. This "Abstraction-Refinement" loop has led to further improvements of this basic idea, driven by the strength of the SAT solver in finding efficient refutations [AM04, AM07]. Through the successive improvements, the SAT solver is brought into play more and more as a deductive reasoning engine. We may speculate about the future: next, perhaps, quantifiers will be supported. Eventually, automated theorem proving may reemerge through this thrust as truly automated deduction inspired by DPLL-style deductive procedures.

The original localization reduction algorithm was implemented in 1992 into the COSPAN model checker [HHK96], when the second author (Kurshan) worked in Bell Labs Research. However, he was barred from publishing the details on account of the utilization of COSPAN as the model checker of FormalCheck [For97]. FormalCheck, a design automation tool, was then under development by Lucent Technologies, the parent company of Bell Labs, for internal use and possible eventual commercial licensing. (FormalCheck was in fact released commercially by Lucent in 1997.) To comply with this restriction, the localization reduction algorithm was first published only in a very general form in [Kur94].

In November 25, 1997, the details of the algorithm were published as US patent no. 5,691,925 entitled "Deriving tractable sub-system for model of larger system". It was issued to R. P. Kurshan and R. H. Hardin, who had worked out many of the details that made the algorithm effective.

Once published as a patent, it might have been allowed to discuss the details publicly. However, there was yet some additional sensitivity surrounding this matter: it was planned to license FormalCheck to be marketed by an EDA company (to be chosen at an auction), and notwithstanding the issuance of the patent, the managers of this plan were against further publication of the algorithm at that time. Indeed, in 1998 FormalCheck was licensed by Lucent to be marketed by Cadence Design Systems. With that, the same reluctance to allow publication passed to Cadence.

A bit anti-climactically, in 2000 the second author finally was allowed to present the details publicly. This was done in a talk at the IBM FV'2000 Seminar in Haifa, Israel (August 15-17, 2000). In the same year, [CGJ$^+$00] had been published. There is a cautionary message in this recounting concerning the relative freedom to publish in an academic institution, compared with even such an exemplary research institution as Bell Labs was at the time.

The overview of the localization reduction algorithm published in [Kur94] could be described as follows.

All properties and the design are described by ω-automata, defined in terms of program variables with their respective assignments. (These variables were implicit in [Kur94], which instead described the algorithm in terms of the underlying Boolean algebra of all variable assignments, in keeping with the presentation style of that book.)

The design can be structured in terms of its *variable dependency graph*: the directed graph whose nodes are the design variables and whose directed edges indicate a dependency of the one variable on the other. Of special interest are the connected components of this graph that consist of directed paths to a variable used to define a property with respect to which the design is to be checked. Together, these components comprise the *fan-in cone* of the property (in the design).

Any part of the design outside the fan-in cone of a property cannot influence the truth of the property on the design (at least for safety properties – for general properties, it is more complicated, but the same general idea applies, after tracing variable dependencies from automata acceptance structures).

Initially, any part of the design outside the fan-in cone of the property to be checked is removed, forming the *pruned design.*

The property is first checked in the *top* abstraction comprised of the null design with all design variables free (unconstrained) – *i.e.*, the property is checked with no design. If the property passes, it is a tautology. Otherwise, there is an error track (involving only the variables used to express the property). Using the greedy routine follow described below, a quick attempt is made to lift the error track to an error track of the full pruned design. This attempt will likely fail (in this initial iteration); the cause of the failure would be a reachable state of the pruned design from which no input can cause a transition to the next state of the error track.

There will be some design variables which, had they been free, could have been assigned values that would allow the next transition of the lifted error track. A heuristically small set of such variables is (somehow) identified. Let's call these *blocking variables.* In the variable dependency graph, they will comprise (connected) paths to the *active* variables – the variables of the current abstraction that carry their respective full assignments, including the variables that define the property being checked. (Initially, the set of active variables consists of just the set of variables that define the property to be checked.) This set of variables: the active variables plus the identified blocking variables, together with their full assignments, forms the next abstraction refinement. With respect to the variable dependency graph, all design variables on the boundary of this refinement are freed *i.e.*, assigned nondeterministically within their respective ranges). Let's call the set of these freed variables the *free fence* of the new abstraction. Although it was not discussed in [Kur94], in the COSPAN implementation, the blocking variables were chosen so as to minimize the size of the resulting free fence, in a manner described below.

The algorithm iterates monotonically through successive refinements until either an error track is successfully lifted to the full pruned design (by follow), or the property passes in some abstraction (or a computer memory limitation is hit).

Effectively, this was the description of the algorithm given in [Kur94].

That in [Kur94] this description was given in terms of the underlying Boolean algebra defined by all possible variable assignments and the ω-automata that define the design and property, may be the reason that even this level of generality was allowed to be published: from that description, the algorithm was not evident to the reviewers. While an elucidation certainly was called for, it could have prevented publication at any level of detail of this important algorithm.

In its initial implementation in COSPAN, the follow routine simply used random simulation to attempt to lift the error track to the full design. This was found to result in an over-all faster localization reduction than alternatives based on symbolic simulation and BDD-based constraint-solving (which often lacked the required capacity). Later, with the advent of more efficient SAT, a SAT-based constraint-solver would have provided a much more efficient and effective follow.

Additionally, in each iteration, the abstraction was further simplified through *variable resizing*, a very simple form of predicate abstraction that reduced variable ranges; and through constant propagation (especially effective when some variables had been resized to constants).

The key to the efficacy of the algorithm, however, was the heuristic used to identify the blocking variables of each iteration, due to R. H. Hardin. Based on Wagner's "Maximal Flow Algorithm" [Wag75], it selected a set of blocking variables that approximately minimized the sum of the logs of the number of possible values of the respective variables in the resulting free fence of the abstraction that resulted from a particular choice of blocking variables. This was solved as a minimal flow problem by dividing each variable into an input ivariable and an output ovariable with a single channel between them having capacity equal to the log of the number of free values for the variable. Each ovariable was joined to the foreign ivariables it depends on with an infinite capacity channel. The active variables are an infinite source of flow to the variables they depend on (so the old free fence is fed by an infinite source), and each state variable ovariable has an infinite-capacity channel to an infinite sink. Thus the flow will be limited by the capacity of certain ivariable-to-ovariable channels internal to original variables. These limiting variables form the new free fence.

The handling of automata acceptance conditions added another layer of complexity that is outside the scope of this paper, but their treatment merely entailed an adjustment to the above algorithm.

3 Counterexample-Guided Abstraction Refinement

The counterexample-guided abstraction refinement framework (CEGAR) was developed at Carnegie Mellon University in the summer of 1999 in the context of Yuan Lu's PhD thesis [Lu00] and presented in mid July 2000 at CAV in Chicago [CGJ$^+$00, CGJ$^+$03], cf. Figure 1. In the same conference, recent Turing award winner Amir Pnueli emphasized the importance of abstraction for model checking in his keynote address *"Abstraction, Composition, Symmetry, and a Little Deduction: The Remedies to State Explosion"* [Pnu00]. Interestingly, Pnueli speculated about "useful additional measures of automation" for these hitherto manual techniques.

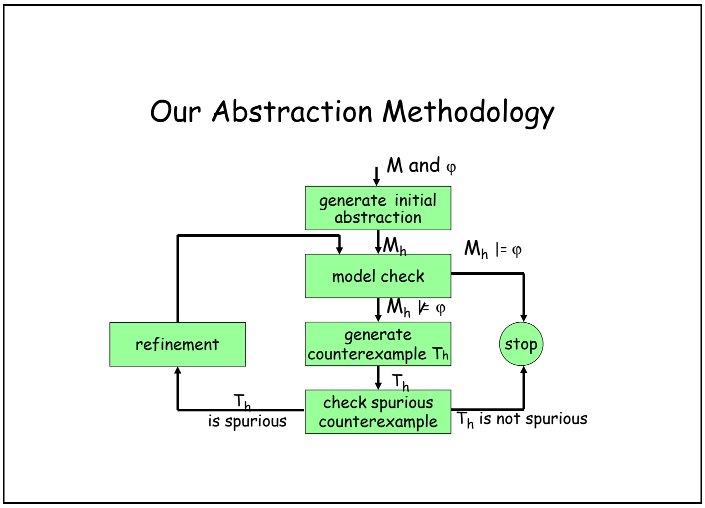
Our Abstraction Methodology
M and φ
generate initial abstraction
M_h
M_h |= φ
model check
M_h ⊭ φ
refinement
generate counterexample T_h
stop
T_h
T_h is spurious
check spurious counterexample
T_h is not spurious

Fig. 1. Slide from CAV 2000

In their approach to the localization reduction, the Lucent management was more successful than in other matters: While the authors were aware of Kurshan's book [Kur94], the localization reduction was unclear to us. Nevertheless we tried to understand it from the book, and it served as an inspiration for CEGAR. The localization reduction was the most prominent paragraph of related work in [CGJ$^+$00].

CEGAR leveraged several threads of research into a new concept for model checking:

– **Existential Abstraction.** CEGAR built on the existing framework of abstraction-based model checking [CGL94]. Given an abstraction function h and an ACTL* property φ, existential abstraction ensures a preservation

$$h(M) \models \varphi \qquad \Longrightarrow \qquad M \models \varphi,$$

i.e., a positive model checking result for property φ on the abstract (small) model $h(M)$ implies truth of the property in the original (larger) system

M. Similarly as in abstract interpretation, the specific abstractions h were manually chosen. The challenge in the manual construction of h is to find the right level of abstraction which reduces the size of the model enough to make $h(M) \models \varphi$ algorithmically tractable, but leaves sufficient precision in the model to avoid false negatives, i.e., situations where $h(M) \not\models \varphi$ and $M \models \varphi$. Note that the possibility of false negatives renders abstract model checking incomplete.

The abstract mapping used in the localization reduction is an instance of existential abstraction, where individual system variables are either unchanged or entirely abstracted away.

- **Predicate Abstraction.** CEGAR made use of predicate abstraction which was introduced by Graf and Saïdi in the context of the PVS theorem prover [GS97]. In predicate abstraction, an abstraction h maps a state s to a formula σ such that $s \models \sigma$. In the seminal paper [GS97], σ is given by $h(s) = \bigwedge_{s \models \psi, \psi \in P} \psi \wedge \bigwedge_{s \models \neg\psi, \psi \in P} \neg\psi$ where P is a finite set of atomic predicates over the program variables. While predicate abstraction nicely fits into abstract model checking [CGL94], it provides two important advantages: First, an abstract state σ can represent an infinite set of concrete states $h^{-1}(\sigma)$. Second, algorithmic reasoning about abstractions can be relegated to decision procedures. Both advantages became important for the development of software model checking.
- **Refinement.** CEGAR extended the idea of counterexample-guided refinement from localization reduction [Kur94] for COSPAN to this more general framework. Counterexample-guided refinement gives an algorithmic alternative to the manual construction of abstraction functions: The model checker starts with a coarse abstraction h. If a spurious counterexample occurs on $h(M)$, the abstraction function is refined as to eliminate the spurious counterexample, and model checking resumes with the refined abstract model, Thus, abstract model checking in combination with abstraction refinement yields an efficient and complete model checking procedure.
- **Symbolic Simulation of Abstract Counterexamples.** CEGAR used symbolic simulation of the abstract counterexample to decide if a counterexample is spurious, and to determine a refined abstraction function. The first implementation of CEGAR was based on the symbolic model checker NuSMV and thus geared to finite state models. Our symbolic simulation algorithm essentially performed a BDD-based forward simulation which either succeeded in a real counterexample, or got stuck in a "failure state" from which a refinement could be determined.

A significant amount of work was devoted to the simplification of the method and also to find generic terminology. The name of the method itself was a topic of repeated discussions. Thus, the paper went through *many* iterations until it reached its final form.

Methodologically, CEGAR has certainly changed our view of counterexamples [CV03]. Before CEGAR, counterexamples were considered important debugging information for the verification engineer, but not a central part of the verification tool chain, and unappealing from a theoretical perspective. With the arrival of CEGAR and bounded model checking [BCCZ99], counterexamples were treated as witnesses of existential temporal specifications, and thus they became a data structure of interest in their own right. Counterexamples are closely related to the path-sensitivity of a model checking analysis; a reconstruction of CEGAR in an abstract interpretation framework is mathematically feasible, but less natural [GQ01].

In fact, the method of [CGJ$^+$00] suffered from a flaw of beauty symptomatic of the situation before 2000. CEGAR was presented for the fragment of ACTL* which admits counterexamples of the form produced by SMV, i.e., finite paths or finite paths leading to a finite loop. It is of course easy to see that many ACTL* specifications such as **AFAX**p require more complex counterexamples, but for a long time this was a blind spot in the model checking literature. The first two decades of model checking research neither produced a precise definition of counterexamples, nor a systematic study of counterexamples for ACTL*.

Both questions were addressed in a follow-up paper at LICS 2002 [CJLV02]. The paper thoroughly investigated the notion of counterexamples – note that the whole system always is a counterexample, yet typically a useless one – and gave a semi-formal definition of counterexamples which is based on simulation but also requires "simplicity" (algorithmic or cognitive simplicity) of counterexamples.

Importantly, the 2002 paper demonstrated the completeness of CEGAR for full branching time logic: First, it was shown that each specification in a large class of logics including ACTL* has a tree-like counterexample. Tree-like counterexamples are structures obtained by recursively gluing together finite paths and finite loops. Then, the paper demonstrated how the CEGAR method of [CGJ$^+$00] and the BDD-based implementation can be extended to full ACTL*.

In two other CEGAR successor papers of 2002 at FMCAD and CAV [CCK$^+$02, CGKS02], CEGAR was studied in a bounded model checking framework. In both papers, a SAT solver is used for the symbolic simulation of a possibly spurious abstract counterexample. If the SAT instance is unsatisfiable, the abstract counterexample is spurious and needs to be refined. The papers use different techniques for determining the refinement: In [CGKS02], ILP (integer linear programming) and machine learning are used to identify important variables which are absent in the abstraction. The second paper [CCK$^+$02] monitors the SAT checking phase in order to analyze the impact of individual variables. Thus, the paper is a direct predecessor of the work by Amla and McMillan [MA03] discussed in Section 4.

CEGAR has found its most important applications in software model checking. Predicate abstraction facilitates the separation of concerns between decision procedures to extract a finite state model from an infinite-state software model, and model checkers to analyze this model. We briefly sketch the main lines of development. SLAM [BR02], BLAST [HJMS02], and MAGIC [CCG$^+$03, Cha05] combined CEGAR with predicate abstraction and decision procedures. SATABS [CKSY05] made use of a bit-precise program presentation and a SAT solver as decision procedure. SLAM was the first tool to apply a CEGAR loop for software, and the first software model checker with significant industrial impact.

The most important development in refinement was the interpolation method by Ken McMillan [McM03] which was first applied to software model checking in collaboration with BLAST [HJMM04] and promises to be a foundation for a systematic approach to refinement in infinite-state systems.

More recently, software model checking has also turned to liveness properties, a topic of continued interest to Amir Pnueli. Amir collaborated with Rybalchenko and Podelski on a method [PR04, PPR05] which was implemented in the Terminator tool [CPR05] using a counterexample-guided abstraction refinement loop.

A good survey on software model checking can be found in [JM09].

4 Automatic Abstraction without Counterexamples

In a stunning advance of the essential *successive approximations* idea, McMillan and Amla presented in effect "counter-example guided refinement without counter-examples" [MA03]. In their algorithm, the successive abstractions are determined not by the counterexamples but by the unSAT proof used to refute falsification in the original model at a given sequential depth. Since this gives a proof of the absence of a counter-example at the given depth, the authors have termed this "proof-based abstraction". Rather than ".. without counter-examples", their method could also be said to be "counter-example guided refinement for *every* counter-example of a given length".

Their method employs a similar abstraction-refinement loop as the previous methods, but uses unSAT clauses in place of the counter-example. First, they perform SAT-based bounded model checking to some depth k in the pruned design. If this fails to generate a counter-example, then they extract the unSAT clauses from the SAT solver. These constitute a proof that no counter-example exists at depth k (or less).

The proof extraction process begins with conflict clause generation derived from the sequence of resolution steps that follow the implication graph. For each generated conflict clause, they record the sequence of clauses that were resolved to produce it. A proof of unsatisfiability follows through a depth-first search that begins at the empty clause and recursively deduces each successive clause in terms of the sequence of clauses that produced it.

Viewing the bounded model checking problem as a set of constraints (initial constraints, transition constraints and final or liveness constraints – for a safety or eventuality property, respectively), they consider the set of these constraints (represented in binary conjunctive normal form) that are used in the proof of unsatisfiability. Any constraint, all of whose clauses are not used in the proof of unsatisfiability, can be removed without affecting the proof. After such removal of constraints, the resulting model is a conservative abstraction of the original model that is also guaranteed to admit of no counter-example of length k (or less).

Next, they perform ordinary (unbounded) model checking on this conservative abstraction. If it passes, then the property is verified for the original model. If it is falsified at some depth k', then they know that $k' > k$, as a counter-example of length k or less has already been ruled out in both the original and abstract model.

Their algorithm now iterates by performing a new bounded model checking run to length k'. This is guaranteed to terminate (or run out of memory), as k is strictly increasing, but cannot exceed the depth of the generated abstract model. When it is equal, the check of the abstract model will verify.

Note that falsification occurs only in the bounded model checking step on the full model, so there can be no issue of a bogus counter-example coming from a check of an abstract model. Conversely, verification occurs only during a check of the abstract model.

While they use SAT-based bounded model checking for the bounded check, they have found that BDD-based model checking is often the most effective type of model checker on the abstract model, as the abstractions are often small.

Note also that unlike the previous methods for which the abstraction refinements grow monotonically, in their procedure, there may be no relationship between successive abstractions. In their procedure, though, the *length* of the generated counter-examples is strictly increasing.

They report that in practice, they have found that when the procedure terminates, k is roughly half the depth of the abstract model.

Acknowledgements. We are thankful to Yuan Lu for comments on Section 3.

References

[AIKY93] Alur, R., Itai, A., Kurshan, R.P., Yannakakis, M.: Timing Verification by Successive Approximation. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 137–150. Springer, Heidelberg (1993); Also Inf. Comput. 118(1), 142–157 (1995)

[AM04] Amla, N., McMillan, K.L.: A Hybrid of Counterexample-Based and Proof-Based Abstraction. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 260–274. Springer, Heidelberg (2004)

[AM07] Amla, N., McMillan, K.L.: Combining Abstraction Refinement and SAT-Based Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 405–419. Springer, Heidelberg (2007)

[BCCZ99] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

[BR02] Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. In: POPL, pp. 1–3 (2002)

[CCG$^+$03] Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: ICSE, pp. 385–395. IEEE Computer Society, Los Alamitos (2003)

[CCK$^+$02] Chauhan, P., Clarke, E.M., Kukula, J.H., Sapra, S., Veith, H., Wang, D.: Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 33–51. Springer, Heidelberg (2002)

[CGJ$^+$00] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)

[CGJ$^+$03] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)

[CGKS02] Clarke, E.M., Gupta, A., Kukula, J.H., Strichman, O.: Sat based abstraction-refinement using ilp and machine learning techniques. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 265–279. Springer, Heidelberg (2002)

[CGL94] Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (1994)

[Cha05] Chaki, S.: A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs. PhD Thesis, Carnegie Mellon University (2005)

[CJLV02] Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: LICS, pp. 19–29. IEEE Computer Society Press, Los Alamitos (2002)

[CKSY05] Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Satabs: Sat-based predicate abstraction for ansi-c. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)

[CPR05] Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)

[CV03] Clarke, E.M., Veith, H.: Counterexamples revisited: Principles, algorithms, applications. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 208–224. Springer, Heidelberg (2004)

[DD01] Das, S., Dill, D.L.: Successive Approximation of Abstract Transition Relations. In: LICS 2001, pp. 51–58. IEEE Computer Society Press, Los Alamitos (2001)

[For97] Lucent's Bell Introduces FormalCheck. Electronic News (April 1997)

[GQ01] Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, p. 356. Springer, Heidelberg (2001)

[GS97] Graf, S., Saıdi, H.: Construction of abstract state graphs with pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

[HHK96] Hardin, R.H., Har'El, Z., Kurshan, R.P.: COSPAN. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 423–427. Springer, Heidelberg (1996)

[HJMM04] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. ACM, New York (2004)

[HJMS02] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)

[JM05] Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)

[JM09] Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. 41(4) (2009)

[Kur94] Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)

[Kur00] Kurshan, R.P.: Program Verification. Notices of the AMS 47(5), 534–545 (2000)

[Lu00] Lu, Y.: Automated Abstraction in Model Checking. PhD Thesis, Carnegie Mellon University (2000)

[MA03] McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)

[McM03] McMillan, K.L.: Interpolation and sat-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)

[Pnu00] Pnueli, A.: Keynote address: Abstraction, composition, symmetry, and a little deduction: The remedies to state explosion. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, p. 1. Springer, Heidelberg (2000)

[PPR05] Pnueli, A., Podelski, A., Rybalchenko, A.: Separating fairness and well-foundedness for the analysis of fair discrete systems. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 124–139. Springer, Heidelberg (2005)

[PR04] Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society, Los Alamitos (2004)

[Wag75] Wagner, H.M.: Principles of Operations Research, pp. 953–958. Prentice Hall, Englewood Cliffs (1975); see Appendix I: Advanced Topics in Network Algorithms