CEDAR: Counter-Example Driven Abstraction Refinement

A Pattern Supporting Formal Verification of Large Systems

Georg Weißenbacher, Wolfgang Herzner
[georg.weissenbacher|wolfgang.herzner]@arcs.ac.at
ARC Seibersdorf research GesmbH

1 Introduction

With the recent advances in information technology, electronic devices continue to permeate our environment and become an indispensable (but more and more *unnoticed*) part of our everyday life. They control the brakes in our cars, guard our houses, or trigger our heartbeat as cardiac pacemakers. The more we rely on these systems, the smaller becomes the step from benefit to threat. Therefore, every precaution has to be taken to rule out unwanted or unexpected behaviour.

Formal Methods like Theorem Proving and Model Checking are rigorous approaches to verify the correctness of software and hardware systems. These methods go far beyond conventional system testing: Model checkers perform an exhaustive exploration of the state space in order to uncover flaws in the system under test. Theorem provers aim to establish a mathematical proof of the correctness of a model.

Both approaches are very ambitious and tend to become overly costly (or even infeasible) with increasing complexity of the system under test. We present a pattern that can be used to enable formal verification tools to handle the complexity of large software or hardware systems. To our knowledge, the first use of this pattern can be observed in Clarke's paper on "Counterexample-guided Abstraction Refinement" [CGJ+00]. Subsequently, it was repeatedly deployed for program verification [BR02b, HJMS03, CCG+03] (where the approach is referred to as "Counterexample-driven Refinement") and in a slightly different manner in the context of automated reasoning [FJOS03, BDS02, BCLZ04].

This pattern is targeted at both experts in formal verification, and people from other problem domains who are interested in learning about standard practices in formal methods. Hopefully, new applications for the CEDAR pattern can be found beyond its original domain of application.

2 Terminology

The pattern we present emerged from the domain of formal methods. Since it is likely that not all readers are familiar with this area, we provide a short summary of the terminology that we use throughout this paper.

Permission is granted to EuroPLoP to make copies for conference use.

Model A model is generally understood as an abstract and theoretical representation of an artefact. A model can be *refined* by increasing precision. We refer to the most precise model in a sequence of refinement steps as the concrete (or original) model, which will in fact denote the artefact itself throughout this paper.

Behaviour The notion of *behaviour of a model* must be understood as a generic term, which can denote valid states, configurations (allowed values of variables), or feasible sequences of state transitions.

Details Details impose restrictions on the behaviour of a model, i.e. adding details to an abstraction reduces the number of its possible behaviours. The omission of details may result in *non-deterministic* behaviour of a model (e.g., if the condition of a conditional statement is removed, either alternative must be considered).

Property A property of a model holds if there is no valid behaviour that violates the property. A property can be understood as partial specification of the behaviour of the model. For example, a property could be that a certain erroneous state cannot be reached, or that a certain sequence of transitions cannot occur.

Abstraction An abstraction denotes a model that has a reduced complexity (e.g., a smaller state space) compared to the original model, but preserves the properties of interest.

Model Checking An exhaustive exploration of the state space of a model with the intention to refute a property. If a violation of the given property is detected, the model checker provides a counterexample (i.e., an explanation of the violation). A *complete* model checking algorithm finds all existing violations. In general, there is no complete model checking algorithm which terminates after a finite number of computation steps for models with an unbounded state space.

Theorem Proving A technique to construct a formal proof of the correctness of a model. Less expressive logics like propositional logic can be handled more or less efficiently, while powerful logics like first- or higher-order logic are in general undecidable.

3 CEDAR: Counterexample-Driven Abstraction Refinement

3.1 Context

You want to apply or develop tools (or methods) for the *formal* verification of properties of large systems (e.g., software modules or hardware components).

3.2 Problem

The 'off-the-shelf' verification technique you decided to use fails to succeed due to the size of the model under examination, i.e. the problem cannot be tackled by immediate application of common verification techniques. How can you improve the approach such that it will be able to handle larger systems?

3.3 Forces

Verification techniques like model checking [EMCGP99] are based on an exhaustive exploration of the (potentially infinite) set of states of a system (or its model). The approach of explicit state enumeration suffers from the so called *state explosion problem* (the size of the state space of a system is exponential in the length of its description). Even methods that use a symbolic representation of states are restricted to relatively small examples. To sum up, automatic verification techniques tend to become less successful with growing complexity of the system under test.

Abstraction is probably the most important technique to cope with the complexity of large systems. An abstract model of a system simulates the original system (i.e., the checked properties are preserved) and is usually much smaller [EMCGP99]. Alas, finding an appropriate abstract model is far from trivial. An unsound abstraction approach invalidates the verification by yielding a model that omits relevant properties of the original system. An abstraction that is too coarse describes properties that are not present in the original system, in which case the verification tool possibly reports invalid flaws (also known as *false positives* or *superfluous counterexamples*).

3.4 Examples

This section presents two examples that illustrate the problems described in the previous section. In Section 3.7, we will explain how CEDAR was applied to solve these problems.

3.4.1 Property Checking for C Programs

Device driver routines that run in the context of the kernel (for reasons of efficiency) can cause significant harm to the operating system. The search for defects in device drivers is tedious. Microsoft certifies a vast number of device drivers from different vendors, and therefore automation of this job seemed very desirable. Device drivers typically consist of several thousand lines of C code, a quantity, which can be pretty hard to handle for model checking tools. However, even for an experienced programmer, a thorough manual examination of a device driver would take much longer.

3.4.2 Satisfiability of First-Order Formulas

Formal verification frequently involves answering the question of whether there is an assignment of values to variables that makes a certain first-order logic formula true. In general, the number of values (or *ground instances*) that must be considered is infinite. Therefore, a naive enumeration of all variable assignments is bound to fail for logics which are more expressive than propositional logic. Symbolic approaches like term rewriting or unification avoid the explicit enumeration of assignments, but they often depend on heuristics, or a considerable amount of user interaction. Therefore, efficiency is still an issue.

3.5 Solution

Assume that you want to determine whether a certain property `p` holds for the system `m` (we will refer to `m` as the *original* or *concrete* model). Follow the abstraction refinement scheme outlined in Figure 1: By applying an abstraction algorithm to the original model `m` you gain

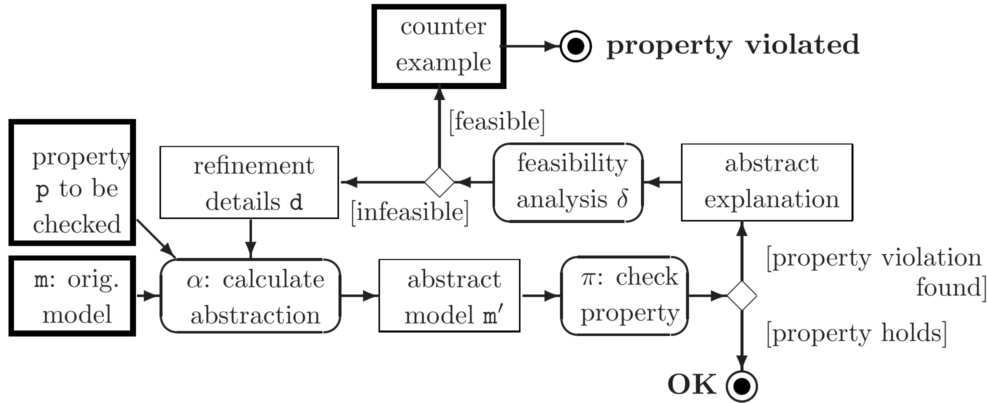

Figure 1: Counterexample-Guided Abstraction Refinement Scheme

an abstract model $\mathtt{m}'$, which still shares the properties of interest with the original model. The reduced complexity of the abstract model $\mathtt{m}'$ makes the application of verification techniques possible that would otherwise be infeasible. All *relevant behaviours* of the original model must be preserved in the abstraction, otherwise the *completeness* of the approach would be forfeited. However, the abstract model may also describe some behaviour that is not present in the original model.

Then, you apply an appropriate verification technique to $\mathtt{m}'$. If it turns out that the property `p` in question holds for $\mathtt{m}'$, you can conclude that `m` is incapable of exposing a behaviour that violates `p`. If it turns out that $\mathtt{m}'$ invalidates the property, you have to determine whether this behaviour is also present in `m`. If this is indeed the case, a feasible counterexample has been found and `p` does not hold for `m`. But it may also happen that this counterexample results from the coarseness of the abstraction. In the latter case, it is necessary to refine the abstraction so that the behaviour that caused the superfluous counterexample is eliminated.

This abstraction refinement scheme is applied repeatedly until either a (feasible) behaviour that violates the property is found, or if it turns out that there is no such behaviour. In certain cases it may also happen that the refinement algorithm yields a model that is too complex to apply the verification technique.

3.6 Structure

CEDAR consists of three basic elements:

1. An abstraction algorithm α, which takes a concrete model `m` and a set of *details* `d`. $\alpha(\mathtt{m}, \mathtt{d})$ yields an abstract model $\mathtt{m}'$ that preserves the details `d`.

2. A property checker π, which determines if $\mathtt{m}'$ violates a property `p` and reports a corresponding behaviour $\mathtt{b} = \pi(\mathtt{m}', \mathtt{p})$ (if such a behaviour exists).

3. A feasibility analysis algorithm δ, which checks if a behaviour `b` provided by π is feasible in the original model. If `b` turns out to be infeasible in `m`, δ provides a set of details $\mathtt{d}' = \delta(\mathtt{m}, \mathtt{b})$ that explain the infeasibility.

α(m,d) yields an abstraction m′ by maintaining only as much detail as specified by d. Obviously, the level of detail must be chosen carefully, since we demand that the resulting abstract model shares certain properties with the original program. No restriction is imposed on how such details can be represented. (Note, that the abstract model may be defined in a language that is less expressive than the formalism used to define the concrete model.) In general, it is a good idea to remove all details that are not somehow related to the property that should be checked (e.g., when checking the reachability of a label, remove all computations that do not influence the control flow). A framework for approximation for program analysis is presented in [CC77]. Predicate abstraction [GS97] has turned out to be a successful abstraction mechanism for several domains. Clarke [CGJ+00] presents an automatic approach to generate abstractions that preserve the model's properties with respect to ACTL* specifications.

The abstraction algorithm α maps the states of the concrete model to states of the abstract domain. A reduction of complexity is achieved by subsuming a set of concrete states by means of a single state in the abstract domain.

The example in Figure 2 illustrates how the concrete states x_1 and x_2 are mapped to an abstract state X, and similarly Y subsumes y_1 and y_2.

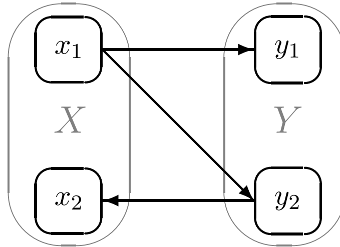

Figure 2: An Example Mapping from Concrete States to Abstract States

The mapping must be conservative in the sense that relations (e.g., valid transitions) between states are preserved. The over-approximation guarantees that the set of behaviours of the abstraction m′ (furtheron denoted as β(m′)) is a superset of behaviours of m, i.e. β(m′) $\supseteq$ β(m). We denote such an abstraction as *sound abstraction*. Note, that one single behaviour b $\in \beta$(m′) potentially subsumes a set of behaviours of m. We say that b $\in \beta$(m′) is feasible in m (simply expressed by b $\in \beta$(m)) if there exists at least one corresponding behaviour that is feasible in m.

In Figure 2, the abstract transition $X \to Y$ summarizes the concrete transitions $x_1 \to y_1$ and $x_1 \to y_2$, and $Y \to X$ corresponds to $y_2 \to x_2$. The behaviour $X \to Y \to X$ is feasible in m, because it maps to $x_1 \to y_2 \to x_2$. However, $Y \to X \to Y$ is feasible only in the abstract model.

The need for a *sound* abstraction algorithm arises from the following considerations: Remember that we want to determine whether a property p holds for a concrete model m, and an immediate application of the property checker π to the concrete model may be infeasible. Provided that the abstraction mechanism is appropriate, it is less costly to compute π(m′, p). If π(m′, p) yields a behaviour b that violates p, we check if b $\in \beta$(m) by computing δ(m, b). If this is the case, then p obviously does not hold for m. If π(m′, p) is unable to find such a behaviour, then p holds for m, because all behaviours of the concrete model were considered.

Now consider the case that $\mathtt{b} \notin \beta(\mathtt{m})$. Obviously, this may happen if the abstraction $\mathtt{m}'$ is too coarse. Therefore, we need a more detailed abstraction $\mathtt{m}''$ that differs from $\mathtt{m}'$ at least so much that $\mathtt{b} \notin \beta(\mathtt{m}'')$ (while still being sound). This can be achieved by finding additional details $\mathtt{d}' = \delta(\mathtt{m}, \mathtt{b})$ and computing $\mathtt{m}'' = \alpha(\mathtt{m}, \mathtt{d} \cup \mathtt{d}')$. Thus, $\mathtt{b}$ will not be reported again by $\pi(\mathtt{m}'', \mathtt{p})$ in the next iteration.

For instance, the reason for the infeasibility of $Y \to X \to Y$ in Figure 2 is that neither y_1 nor y_2 can be reached from x_2. Therefore, the abstraction can be refined by partitioning X.

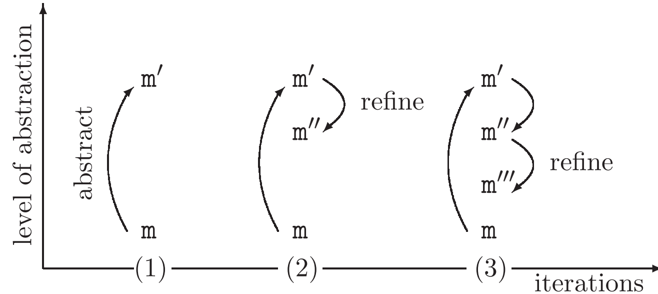

Figure 3: Iterative Abstraction Refinement

By repeatedly following the steps described above we generate a sequence of abstractions until either a violation of $\mathtt{p}$ in $\mathtt{m}$ is found, or until we have evidence that $\mathtt{p}$ holds in $\mathtt{m}$. Note that this sequence $[\mathtt{m}', \mathtt{m}'', \ldots, \mathtt{m}^i, \mathtt{m}^{i+1}, \ldots]$ is potentially infinite, i.e. CEDAR may be non-terminating (The reason for this is that property checking is in general undecideable). It might therefore be necessary to introduce an upper bound n for the number of iterations. $\alpha(\mathtt{m}, \emptyset)$ is the most abstract element in this sequence of abstractions. *If* $\mathtt{m}$ is an element of this sequence, it is always the final (most detailed) element, otherwise the sequence *converges* to $\mathtt{m}$. Figure 3 illustrates how the abstractions $\mathtt{m}^i$ are refined step by step.

3.7 Examples Resolved

3.7.1 Checking Properties of C Programs with SLAM

The SLAM toolkit [BR02b] was developed to check temporal safety properties of C programs. The tool has been successfully deployed on Windows XP device drivers. SLAM automatically examines if the driver makes correct use of the kernel API, e.g., if locks are acquired and released in the correct order. In the SLAM process, a sequential C program acts as the original model $\mathtt{m}$. A safety property $\mathtt{p}$ is specified by instrumenting the C program $\mathtt{m}$ such that a unique label `ERROR` is *reachable* in $\mathtt{m}$ if (and only if) $\mathtt{m}$ does not satisfy $\mathtt{p}$.

The abstract models $\mathtt{m}^i$ are represented in terms of *Boolean Programs* [BR00b]. Boolean Programs are similar to C programs, apart from the fact that they may only contain boolean variables. Reachability of labels in Boolean Programs is decideable.

The SLAM toolkit is comprised of three basic tools:

- C2BP, a tool which transforms the original C program $\mathtt{m}$ into a Boolean Program. This tool implements the abstraction algorithm α. C2BP uses a mechanism called Predicate Abstraction [GS97, BMMR01, BMR02] to generate an abstraction with respect to a finite set of predicates which range over the variables in the C program $\mathtt{m}$. Each variable

in the Boolean Program corresponds to one predicate. The predicates represent the details `d`. Assume that `d` consists of a single predicate $(i < 5)$. The abstract state in which $(i < 5)$ evaluates to `true` subsumes all concrete states in which i is less than 5. Conversely, when $(i < 5)$ becomes `false`, the corresponding set of concrete states consists of all states in which i is greater than or equal to 5.

Now let us assume that `m` contains a conditional statement `if` $(i < 6)$ `then ...else ...`. If $(i < 5)$ evaluates to `true` in the corresponding abstract program $\mathtt{m}'$ at the time when the `if` statement is encountered, only the `then` branch must be considered, since $(i < 5)$ implies $(i < 6)$. However, if $(i < 5)$ is `false`, both branches must be considered by the model checker, since no assumptions can be made on the exact value of i.

An assignment statement `i = i-1;` is abstracted as follows: If $(i < 5)$ was `true` before the execution of this statement, it must remain `true` afterwards. However, if the predicate evaluated to `false` before the assignment statement was encountered, its value after the assignment statement is unknown. A detailed description of the abstraction process (which is based on the theory of abstract interpretation [CC77]) is given in [BMMR01].

- Bebop [BR00a], a model checker which performs a reachability analysis of Boolean Programs. It uses a symbolic representation to efficiently represent reachable states at each program point. If an erroneous state is reachable, Bebop reports an execution trace $\mathtt{b} = \pi(\mathtt{m}', \mathtt{p})$, which leads to this state.

- Newton [BR02a], a tool that discovers predicates for the refinement. It analyses the feasibility of `b` by performing a symbolic interpretation of `b` in `m`. Since `b` describes only a single behaviour of `m`, only a small subset of the state space has to be considered during the feasibility analysis. An execution trace is infeasible in `m` if an assumption is violated. Assume that `b` passes the conditional statement `if` $(i < 6)$ mentioned in the example above and demands that the `then` branch is executed. If the symbolic interpretation algorithm reveals that i cannot be smaller than 6 in the given execution trace, the assumption $(i < 6)$ is violated. Slam would proceed by calculating a new abstraction $\mathtt{m}'' = \alpha(\mathtt{m}, \mathtt{d} \cup \{(i < 6)\})$.

3.7.2 Satisfiability of First-Order Formulas

The Verifun tool presented in [FJOS03] incrementally translates a first-order formula to a propositional formula. Numerous algorithms (e.g., [MMZ+01, MSS99, Zha97]) that target Boolean Satisfiability (SAT), the problem of determining a satisfying variable assignment for a propositional formula, have been proposed and implemented.

In order to adhere to the terminology established in Section 3.5 the problem has to be slightly reformulated. A feasible assignment `b` for a first-order logic formula would be a witness to its satisfiability rather than a counterexample. However, `b` constitutes a counterexample to the invalidity of the formula. Therefore, in order to determine if a formula is satisfiable, the property we have to ask for is if `m` is invalid. Alternatively, one can simply interpret the counterexample as a witness.

- The abstraction algorithm α replaces each distinct atomic formula φ with a new propositional variable A_φ. The result is a purely propositional formula. The abstraction is sound in the sense that, given any satisfying assignment for the first-order logic formula

$\mathtt{m}$, a satisfying assignment for $\mathtt{m}'$ can be obtained by assigning to each A_φ the truth value of the interpretation of the corresponding atomic formula φ. The converse does not hold in general [FJOS03]. Formula 1 below gives an example (based on [FJOS03]) of how $\mathtt{m}' \equiv A_1 \wedge (\neg A_2 \vee A_3)$ is generated from $\mathtt{m} \equiv (a = b) \wedge (\neg(f(a) = f(b)) \vee (b = c))$:

$$\overbrace{(a = b)}^{A_1} \wedge (\neg\overbrace{(f(a) = f(b))}^{A_2}) \vee \overbrace{b = c)}^{A_3} \qquad (1)$$

- A SAT solver takes the role of the property checker π. It returns a satisfying assignment $\mathtt{b}$ for $\mathtt{m}'$ (if such an assignment exists). In the context of the example given above, assume that $\mathtt{b} \equiv [A_1 = \mathtt{true}, A_2 = \mathtt{false}, A_3 = \mathtt{false}]$.
- $\mathtt{b}$ can be converted into a first-order formula by substituting φ for each corresponding A_φ. This transformation yields a conjunction of literals. In VERIFUN, a decision procedure for the theory of equality with uninterpreted function symbols is used to check if these literals are consistent. If this is the case, a valid assignment for the original formula has been found and $\mathtt{m}$ is satisfiable. Otherwise, the decision procedure reports a set of literals that contribute to the inconsistency. A refined model $\mathtt{m}''$ is constructed by augmenting $\mathtt{m}'$ with an additional clause $\mathtt{d}'$, which prevents these literals from being assigned inconsistently again in the next iteration.

  In the case of our example, the decision procedure would determine that the literals $a = b$ and $\neg(f(a) = f(b))$ are inconsistent, and that there is no point in considering any $\mathtt{b}$ with $A_1 = \mathtt{true}$ and $A_2 = \mathtt{false}$ in subsequent iterations. Therefore, $\mathtt{d}' \equiv \neg A_1 \vee A_2$ is used to refine $\mathtt{m}'$, yielding $\mathtt{m}'' \equiv A_1 \wedge (\neg A_2 \vee A_3) \wedge (\neg A_1 \vee A_2)$.

3.8 Consequences

3.8.1 Benefits

- CEDAR makes it feasible to solve extremely resource intensive problems like formal verification, which would otherwise stay beyond available memory sizes and computing times on todays equipment.
- The approach yields a counterexample (or a *witness*, depending on the point of view) that serves as an explanation for the violation of the property in question.
- Instead of burdening the user with the identification of false positives, superfluous counterexamples are used to refine the abstraction.
- CEDAR helps to avoid errors during abstraction steps because it supports the identification of details that must be preserved in the abstraction. Additionally, automatic abstraction methods are encouraged, and in the ideal case, the developer is not forced to provide an abstraction of the system under test at all.

3.8.2 Liabilities

- Finding an appropriate abstraction mechanism requires a great deal of creativity. The problem is similar to the problem of finding equivalence classes for test cases.

- Depending on the abstraction algorithm and on the structure of the model under test, it may happen that in the nth iteration $\mathtt{m}^n$ becomes too complex to apply the chosen verification method. In this special case (but not in general), there is no benefit in applying CEDAR.

- Undecidability cannot be overcome. In general, the problem of checking whether `p` holds for the concrete model `m` is *undecidable*. This problem is not specific to the pattern we presented, but common to all verification techniques. It should therefore not be regarded as a disadvantage of the CEDAR pattern, but simply as a fact that cannot be changed.

- Though the abstraction is done automatically, it might still be necessary to provide a harness and stubs in order to establish well defined boundaries for the system under test (e.g., typically you do not want to consider system libraries when you apply a model checker to your system under test). CEDAR does not provide any aid in generating such a harness. However, a test harness (also known as test driver) is also necessary for conventional unit testing.

3.9 Known Uses

The methodology presented in [CGJ+00] has been implemented in the NuSMV [CCGR00] model checker, which targets the verification of industrial designs (e.g., processors). Both the Blast tool [HJMS03] as well as the Magic tool [CCG+03] are used to verify safety properties of C programs, following the same principles as Slam. Blast features an incremental abstraction algorithm. Rather than performing the abstraction from scratch, the information gained during the construction of $\mathtt{m}'$ is reused for the refinement $\mathtt{m}''$.

The tools Zapato [BCLZ04] and CVC [BDS02] use an approach similar to Verifun [FJOS03]. Zapato is particularly interesting, since it is used by C2bp (see Section 3.7.1) for the computation of the predicate abstraction. Therefore, the Slam toolkit constitutes a recursive usage of the CEDAR pattern. The ASAP tool [KOSS04] checks the satisfiability of quantifier free Presburger arithmetic by computing an abstraction in which integer variables are only allowed to range over a bounded domain.

The Salsa tool [Bha01], a decision procedure based invariant checker for the SAL specification language, deploys CEDAR in a less automated way. Checking the feasibility of counterexamples as well as providing additional details in terms of auxiliary lemmas is in the responsibility of the user.

3.10 Related Patterns

To the best of our knowledge, there are no other patterns targeting formal verification and automatic program analysis.

Acknowledgements

We would like to express our gratitude to our shepherd Lars Grunske, for his inspiring comments, which, as we believe, resulted in a significant improvement of the readability of this paper. Furthermore we want to thank our colleagues Brian Sallans and Gerhard Russ for thorough proofreading and commenting on this paper.

References

[BCLZ04] Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: automatic theorem proving for predicate abstraction refinement. In *CAV '04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 457–461. Springer-Verlag, 2004.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 236–249. Springer-Verlag, 2002.

[Bha01] Ramesh Bharadwaj. Analysis of agent-based systems using decision procedures. In *FAABS '00: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers*, pages 298–299. Springer-Verlag, 2001.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213. ACM Press, 2001.

[BMR02] Thomas Ball, Todd Millstein, and Sriram K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR 2001-10, Microsoft Research, June 2002.

[BR00a] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130. Springer-Verlag, 2000.

[BR00b] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR 2000-14, Microsoft Research, February 2000.

[BR02a] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR 2002-09, Microsoft Research, January 2002.

[BR02b] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 2002: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM Press, 2002.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CCG+03] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 385–395. IEEE Computer Society, 2003.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proccedings of the 12th International Conference on Computer Aided Verification*, pages 154–169. Springer-Verlag, 2000.

[EMCGP99] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 1999.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *CAV '03: Proceedings of the 15th International Conference on Computer Aided Verification*, pages 355–367. Springer-Verlag, 2003.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83. Springer-Verlag, July 1997.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Springer-Verlag, 2003.

[KOSS04] Daniel Kröning, Joël Ouaknine, Sanjit Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *CAV '04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 308–320. Springer-Verlag, 2004.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535. ACM Press, 2001.

[MSS99] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[Zha97] Hantao Zhang. Sato: An efficient propositional prover. In *CADE-14: Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275. Springer-Verlag, 1997.