Decision Procedures for the Grand Challenge

Daniel Kroening

Computer Systems Institute
ETH Zürich

Abstract. The *Verifying Compiler* checks the correctness of the program it compiles. The workhorse of such a tool is the reasoning engine, which decides validity of formulae in a suitably chosen logic. This paper discusses possible choices for this logic, and how to solve the resulting problems.

1 Introduction

The solution to the *Grand Challenge* proposed by Tony Hoare [1] is close to millions of programmers' daydream: a compiler that automatically detects all the bugs in the code.

More realistically, the goal is to prove or refute assertions given together with the program. Writing assertions is common practice. It will certainly remain difficult to write a specification that is strong enough to capture the designer's intent, but leaving this problem aside, just checking what we are able to specify would be of tremendous usefulness already.

The way these assertions are specified is intentionally left open; this may range from simplistic `assert()` statements inserted into the code to a formulae given in a temporal logic like LTL to even another higher-level program, which serves as specification. In general, it is to be expected that the specification or the assertions themselves will not be strong enough to serve as inductive invariants for loop constructs. Part of the challenge, therefore, is to strengthen the property to allow reasoning about the loops.

Manifold methods have been proposed to address this challenge. A sign for the feasibility of the task is the success of formal verification tools in the hardware industry. Introduced in 1981, *Model Checking* [2, 3] is one of the most commonly used formal verification technique in a commercial setting. It's main advantage is automation. In contrast to interactive theorem proving, no manual effort is required. However, it suffers from the state explosion problem. In case of BDD-based symbolic model checking this problem manifests itself in the form of unmanageably large BDDs [4].

This problem is partly addressed by a formal verification technique called *Bounded Model Checking* (BMC) [5]. In BMC, the transition relation for a complex design and its specification are jointly unwound to obtain a formula, which is then checked for satisfiability. This process terminates when the length of the potential counterexample exceeds its completeness threshold (i.e., is sufficiently long to ensure that no counterexample exists [6]) or when the SAT procedure exceeds its time or memory bounds. BMC has been used successfully to find subtle errors in very large industrial circuits [7, 8].

BMC has recently been adopted to software verification as well. CBMC [9] unwinds sequential ANSI-C programs, flattens the resulting bit-vector logic formula, and

passes the resulting propositional formula to a SAT-solver. TCBMC, developed at IBM Research, is a version of CBMC extended with support for threaded programs [10]. Saturn [11] and F-SOFT [12] implement similar algorithms. An application of BMC to web applications is reported in [13].

The disadvantage of BMC is that it is typically only applicable for refutation; the completeness threshold [6] is too large for most practical instances. The goal of the *Verifying Compiler*, however, is verification, and not refutation. In industrial practice, the principal method for proving properties is *abstraction*. Abstraction techniques reduce the state space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system.

Predicate abstraction [14, 15] is one of the most popular and widely applied methods for systematic abstraction of programs. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state system that is an abstraction of the original system with respect to a set of predicates.

The abstraction refinement process using predicate abstraction has been promoted by the success of the SLAM project at Microsoft Research [16]. One starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the *abstract-verify-refine* paradigm and depend on the abstraction and refinement techniques used.

The workhorse of both BMC and predicate abstraction is the reasoning engine, which decides validity of formulae in a suitably chosen logic. This paper discusses possible choices for this logic, and how to solve the resulting problems.

2 Decision Procedures for Program Verification

2.1 Existing Approaches

Almost all program verification engines, such as symbolic model checkers and advanced static checking tools, employ automatic theorem provers for symbolic reasoning. For example, the static checkers ESCJAVA [17] and BOOGIE [18] use the Simplify [19] theorem prover to verify user-supplied invariants.

The SLAM [20–25] software model-checker uses ZAPATO [26] for symbolic simulation of C programs. The BLAST [27] and MAGIC [28] tools use Simplify for abstraction, simulation and refinement. Other examples include the Invest [29] tool, which uses the PVS [30] theorem prover. Further decision procedures used in program verification are CVC-Lite [31], ICS [32] and Verifun [33].

However, the fit between the program analyzer and the theorem prover is not always ideal. The problem is that the theorem provers are typically geared towards efficiency in the mathematical theories, such as linear arithmetic over the integers. In reality, program analyzers rarely need reasoning for unbounded integers. Linearity can also be too

limiting in some cases. Moreover, because linear arithmetic over the integers is not a convex theory (a restriction imposed by the Nelson-Oppen and Shostak theory combination frameworks), the real numbers are often used instead. Program analyzers, however, need reasoning for the reals even less than they do for the integers.

The program analyzers must consider a number of issues that are not easily mapped into the logics supported by the theorem provers. These issues include pointers, pointer arithmetic, structures, unions, and the potential relationship between these features.

In [34], we proposed the use of propositional SAT-solvers as a reasoning engine for program verification. The astonishing progress SAT solvers made in the past few years is given in [1] as a reason why the grand challenge is feasible today. Solvers such as ZChaff [35] can now solve many instances with hundreds of thousands of variables and millions of clauses. The arithmetic operators in the formula are replaced by corresponding circuits. The resulting net-list is converted into CNF and passed to a SAT solver. This allows supporting all operators as defined in the ANSI-C standard.

In [36], we report experimental results that quantify the impact of replacing ZAPATO, a decision procedure for integers, with Cogent, a decision procedure built using a SAT solver: The increased precision of Cogent improves the performance of SLAM, while the support for bit-level operators resulted in the discovery of a previously unknown bug in a Windows device driver.

This approach is currently state-of-the-art for deciding validity of formulae in a logic supporting bit-vector operators. It is implemented by Cogent and CVC-Lite, while ICS is still using BDDs to reason about this logic.

2.2 Future Work

The existing approach is clearly not satisfying:

1. First of all, the word-level information about the variables is lost when splitting into bits. A solver exploiting this structure is highly desirable. Word-level SAT-solvers (sometimes called circuit-level SAT solvers) attempt to address this problems, but provide only a very small subset of the required logic. In order to compute predicate images or to perform a fixed-point computation, we need to solve a quantification (or projection) problem, not a decision problem, which is typically considered to be harder than the decision problem. We describe a proof-based approach to perform an approximative existential quantification of formulae in bit-vector logic at the word-level in section 3.
2. Second, the logic supported by this approach is still not sufficient. A major goal of a *Verifying Compiler* is to show pointer-safety. In the presence of dynamic data structures, this requires support for a logic such as separation logic [37]. The combination of such a non-standard logic with bit-vector logic in a joint efficient decision procedure is a challenging problem.
3. Programs involving complex data structures will certainly require formulae that use quantifiers, e.g., to quantify over array indices. Due to the high complexity of these decision problems, there are currently no practical decision procedures available. The progress solvers for QBF (quantified boolean formulae) is making is encouraging, and promises to allow new applications just as the progress of SAT-solvers did.

3 Word-Level Reasoning for Bit-Vectors

3.1 Encoding Decision Problems into Propositional Logic

SAT solvers have become an integral part of all modern decision procedures. There are two different ways to compute an encoding of a decision problem ϕ into propositional logic. In both cases, the propositional part ϕ_{enc} of the formula is converted into CNF first.

Definition 1. *Let ϕ denote a formula. The set of all atoms in ϕ that are not Boolean identifiers is denoted by $\mathcal{A}(\phi)$. The* Propositional Encoding *ϕ_{enc} of a bit-vector formula ϕ is obtained by replacing all atoms $a \in \mathcal{A}(\phi)$ by fresh Boolean identifiers $e_1, \ldots, e_\nu$, where $\nu = |\mathcal{A}(\phi)|$. The atom replaced by e_i is denoted by $\mathcal{A}(e_i)$.*

As an example, the propositional encoding of $\phi = (x = y) \wedge ((a \oplus b = c) \vee (x \neq y))$ is $e_1 \wedge (e_2 \vee \neg e_1)$, and $\mathcal{A}(\phi) = \{x = y, a \oplus b = c\}$.

We denote the vector of the variables $E = \{e_1, \ldots, e_\nu\}$ by $\overline{e}$. Furthermore, let $\psi_a(e)$ denote the atom a with polarity p:

$$\psi_a(e) := \begin{cases} a & : p \\ \neg a & : \text{otherwise} \end{cases} \tag{1}$$

Lazy vs. Eager Encodings Linear-time algorithms for computing CNF for ϕ_{enc} are well-known [38]. All decision procedures transform ϕ_{enc} into CNF this way. The algorithms differ in how the non-propositional part is handled.

The vector of variables $\overline{e} : \mathcal{A}(\phi) \longrightarrow \{\mathsf{true}, \mathsf{false}\}$ as defined above denotes a truth assignment to the atoms in ϕ. Let $\Psi_{\mathcal{A}(\phi)}(\overline{e})$ denote the conjunction of the atoms $a_i \in \mathcal{A}(\phi)$ where the a_i are in the polarity given by $\psi(a_i)$:

$$\Psi_{\mathcal{A}(\phi)}(\overline{e}) := \bigwedge_{i=1}^{\nu} \psi_{a_i}(e_i) \tag{2}$$

An *Eager Encoding* considers all possible truth assignments $\overline{e}$ before invoking the SAT solver, and computes a Boolean constraint $\phi_E(\overline{e})$ such that

$$\phi_E(\overline{e}) \iff \Psi_{\mathcal{A}(\phi)}(\overline{e}) \tag{3}$$

The number of cases considered while building ϕ_E can often be dramatically reduced by exploiting the polarity information of a, i.e., whether a appears in negated form or without negation in the negation normal form (NNF) of ϕ. After computing ϕ_E, ϕ_E is conjoined with ϕ_{enc}, and passed to a SAT solver. A prominent example of a decision procedure implemented using an eager encoding is UCLID [39].

A *Lazy Encoding* means that a series of encodings ϕ_L^1, ϕ_L^2 and so on with $\phi \Longrightarrow \phi_L^i$ is built. Most tools implementing a lazy encoding start off with $\phi_L^1 = \phi_{enc}$. In each iteration, ϕ_L^i is passed to the SAT solver. If the SAT solver determines ϕ_L^i to be unsatisfiable, so is ϕ. If the SAT solver determines ϕ_L^i to be satisfiable, it also provides a satisfying assignment, and thus, an assignment $\overline{e}^i$ to $\mathcal{A}(\phi)$.

The algorithm proceeds by checking if $\Psi_{\mathcal{A}\phi}(\overline{e}^i)$ is satisfiable. If so, ϕ is satisfiable, and the algorithm terminates. If not so, a subset of the atoms $\mathcal{A}' \subseteq \mathcal{A}(\phi)$ is determined, which is already unsatisfiable under $\overline{e}^i$. The algorithm builds a *blocking clause* b, which prohibits this truth assignment to $\mathcal{A}'$. The next encoding ϕ_L^{i+1} is $\phi_L^i \wedge b$. Since the formula becomes only stronger, the algorithm can be tightly integrated into one SAT-solver run, which preserves the learning done in prior iterations.

Among others, CVC-Lite [31] implements a lazy encoding of integer linear arithmetic. The decision problem for the conjunction $\Psi_{\mathcal{A}\phi}(\overline{e}^i)$ is solved using the Omega test, which is described in the next section.

3.2 Encodings from Proofs

A proof is a sequence of transformations of facts. The transformations follow specific rules, i.e., proof rules, which are usually derived from an axiomatization of the logic at hand. A proof of a formula ϕ in a particular logic can be used to obtain another formula ϕ_P in propositional logic that is valid if and only if the original formula is valid, i.e., $\phi \iff \phi_P$. Let $\mathscr{F}$ denote the set of facts used in the proof.

Given a proof of ϕ, a propositional encoding of ϕ can be obtained as follows:

1. Assign a fresh propositional variable v_f to each fact $f \in \mathscr{F}$ that occurs anywhere in the proof.
2. For each proof step i, generate a constraint c_i that captures the dependencies between the facts. As an example, the derivation

$$\frac{A, B}{C}$$

   with variables v_A, v_B, v_C for the facts A, B, and C generates the constraint $(v_A \wedge v_B) \longrightarrow v_C$.
3. The formula ϕ_P is obtained by conjoining the constraints:

$$\phi_P := \bigwedge_i c_i$$

However, the generation of such the proof is often difficult to begin with. In particular, it often suffers from a blowup due to case-splitting caused by the Boolean structure present in ϕ. This is addressed by a technique introduced by Strichman in [40]. His paper describes an eager encoding of linear arithmetic on both real numbers and integers into propositional logic using the Fourier-Motzkin transformation for the reals and the Omega-Test [41] for the integers.

The idea of [40] is applicable to any proof-generating decision-procedure:

- All atoms $\mathcal{A}(\phi)$ are passed to the prover *completely disregarding the Boolean structure* of ϕ, i.e., as if they were conjoined.
- For facts f that are also atoms assign $v_f := e_f$.
- The prover must be modified to obtain *all* possible proofs, i.e., must not terminate even if the empty clause is resolved.

Since the formula that is passed to the prover does not contain any propositional structure, obtaining a proof is considerably simplified. The formula ϕ_P obtained from the proof as described above is then conjoined with the propositional encoding ϕ_{enc}. The conjunction of both is equi-satisfiable with ϕ. As $\phi_P \wedge \phi_{enc}$ is purely propositional, it can be solved by an efficient propositional SAT-solver.

3.3 Existential Abstraction

Let S denote the set of concrete states, and $R(x, x')$ denote the concrete transition relation. As an example, consider the basic block

```
i++;
j=i;
```

We use $x.v$ to denote the value of the variable v in state x. The transition relation corresponding to this basic block is then $x'.i = x.i + 1 \wedge x'.j = x'.i$.

Let $\Pi = \{\pi_1, \ldots, \pi_n\}$ denote the set of predicates. The abstraction function $\alpha(x)$ maps a concrete state $x \in S$ to an abstract state $\hat{x} \in \{\mathsf{true}, \mathsf{false}\}^n$:

$$\alpha(x) := (\pi_1(x), \ldots, \pi_n(x))$$

When computing an existential abstraction, the abstract model can make a transition from an abstract state $\hat{x}$ to $\hat{x}'$ iff there is a transition from x to x' in the concrete model and x is abstracted to $\hat{x}$ and x' is abstracted to $\hat{x}'$. Formally, the abstract transition relation is denoted by $\hat{R}$.

$$\hat{R} := \{(\hat{x}, \hat{x}') \mid \exists x, x' \in S : R(x, x') \wedge \alpha(x) = \hat{x} \wedge \alpha(x') = \hat{x}'\} \tag{4}$$

$\hat{R}$ is also called the image of the predicates Π over R. The formula on the right hand side can be transformed into CNF by replacing the bit-vector arithmetic operators in R and α by arithmetic circuits. Due to the quantification over the abstract states this corresponds to an all-SAT instance. Solving such instances is usually exponential in n.

As an alternative, $\hat{R}$ can be computed using a proof. The facts given to the prover are:

1. All the predicates evaluated over state x, i.e., $\pi_i(x)$,
2. all the predicates evaluated over state x', i.e., $\pi_i(x')$,
3. the atoms in transition relation $R(x, x')$.

We then obtain ϕ_B as described in section 3.2. Both ϕ_B and ϕ_{enc} contain fresh propositional variables for the atoms $\mathcal{A}(R)$ in R, for the predicates Π over x and x', and for the facts $f \in \mathscr{F}$ found during the derivation. Let V_R denote the set of propositional variables corresponding to atoms in R that are not predicates, and let V_F denote the set of propositional variables corresponding to facts $f \in \mathscr{F}$ that are not predicates.

The propositional variables that do not correspond to predicates are quantified existentially to obtain the predicate image. Let $\overline{v}_R$ denote the vector of variables in V_R, let

$\overline{v}_F$ denote the vector of variables in V_F, and let $\mu_R = |V_R|$ and $\mu_F = |V_F|$ denote the number of such variables.

$$\hat{R} := \{(\hat{x}, \hat{x}') \mid \exists \overline{v}_R \in \{0,1\}^{\mu_R}, \overline{v}_F \in \{0,1\}^{\mu_F} : \phi_{enc}(\hat{x}, \hat{x}', v_R) \wedge \phi_P(\hat{x}, \hat{x}', v_F)\} \tag{5}$$

Thus, we replaced the existential quantification of concrete program variables $x, x' \in S^2$ by an existential quantification of $\mu_R + \mu_F$ Boolean variables. The authors of [42] report experiments in which this quantification is actually performed by means of either BDDs or the SAT-engine of [34].

The authors of [43] use BDDs to obtain all cubes over the variables in V_F, and then enumerate these cubes. This operation is again worst-case exponential.

As motivated above, reasoning for integers is a bad fit for system-level software, and basically useless to prove properties of hardware. We would therefore like a proof-based method for a bit-vector logic. The main challenge is that any axiomatization for a reasonably rich logic permits way too many ways of proving the same fact, an the procedure as described above relies on enumerating *all* proofs.

We therefore propose to sacrifice precision in order to be able to reason about bit-vectors, and compute an over-approximation of $\hat{R}$. This is a commonly applied technique, e.g., used by SLAM and BLAST. If this over-approximation results in a spurious transition, it can be refined by any of the existing refinement methods, e.g., based on UNSAT cores as in [44] or based on interpolants as in [45].

One trivial way to obtain an inexpensive over-approximation of $\hat{R}$ is, e.g., bounding the depth of proofs. Future research could, for example, focus on better proof-guiding heuristics.

Example Assume we have, among others, the following derivation rules:

$$(a|b)\&b == b \tag{6}$$

$$\frac{b\&c == 0}{(a|b)\&c == a\&c} \tag{7}$$

The predicates we consider are $\pi_1 == (x\&1 = 0)$ and $\pi_2 == (x\&2 = 0)$, and the statement to be executed is `x|=2;`.

The facts passed to the prover are $x\&1 = 0$, $x\&2 = 0$, $x'\&1 = 0$, $x'\&2 = 0$, and $x' = x|2$. Figure 1 shows a derivation on the left hand side and on the right hand side the same derivation tree in which the atoms are replaced by their propositional variables. The derivation results in the constraint $(\pi_2' \longrightarrow v_1) \wedge (v_1 \longrightarrow \mathsf{F})$, which is equivalent to $\neg\pi_2'$. Figure 2 shows a derivation that ends in an existing atom π_1 rather than F. The constraint generated is equivalent to $\pi_1' \longrightarrow \pi_1$.

4 Conclusion

Program verification engines rely on decision procedures. However, despite of many years of research in this area, the available decision procedures are not yet geared towards program analysis. Program analysis requires a logic with many features commonly not found in today's decision procedures, such as bit-vector operators, and ways to handle structs, unions, and pointers, e.g., separation logic.

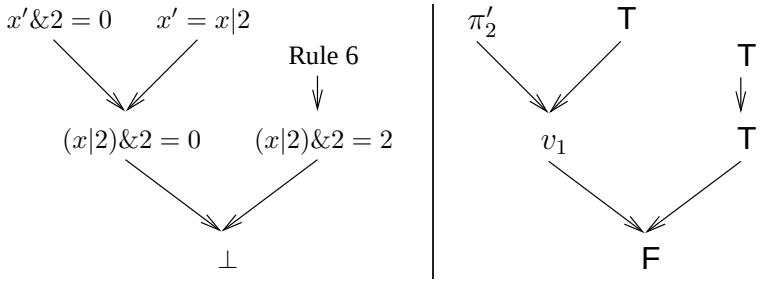

Fig. 1. Derivation of constraints for π_2'

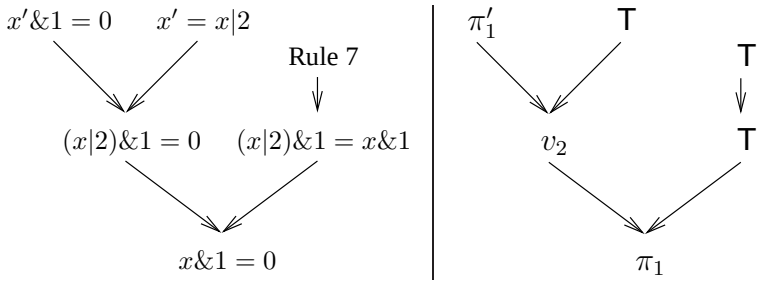

Fig. 2. Derivation of constraints for π_1'

The current state-of-the-art for deciding bit-vector logic is an ad-hoc approach using propositional SAT-solvers. Efficient Decision procedures that support a logic as needed for program analysis is an open problem that has to be solved to succeed in the grand challenge.

References

1. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM **50** (2003) 63–69
2. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
3. Clarke, E.M., Emerson, E.A.: Synthesis of synchronization skeletons for branching time temporal logic. In: Logic of Programs: Workshop. Volume 131 of LNCS. Springer-Verlag (1981)
4. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98** (1992) 142–170
5. Biere, A., Cimatti, A., Clarke, E., Yhu, Y.: Symbolic model checking without BDDs. In: TACAS. (1999) 193–207
6. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In Zuck, L., Attie, P., Cortesi, A., Mukhopadhyay, S., eds.: 4th International Conference on Verification, Model Checking, and Abstract Interpretation. Volume 2575 of Lecture Notes in Computer Science., Springer Verlag (2003) 298–309

7. Shtrichman, O.: Tuning SAT checkers for bounded model checking. In Emerson, E., Sistla, A., eds.: Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000). Lecture Notes in Computer Science, Springer Verlag (2000) 480–494
8. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of bounded model checking at an industrial setting. In Berry, G., Comon, H., Finkel, A., eds.: Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001). Number 2102 in Lecture Notes in Computer Science, Springer Verlag (2001) 436–453
9. Clarke, E., Kroening, D.: Hardware verification using ANSI-C programs as a reference. In: Proceedings of ASP-DAC 2003, IEEE Computer Society Press (2003) 308–311
10. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: International Conference on Computer Aided Verification (CAV). Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 82–97
11. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In Palsberg, J., Abadi, M., eds.: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM (2005) 351–363
12. Ivančić, F., Yang, Z., Ganai, M., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: Software verification platform. In: International Conference on Computer Aided Verification (CAV). Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 301–306
13. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Verifying web applications using bounded model checking. In: Dependable Systems and Networks (DSN), IEEE Computer Society (2004) 199–208
14. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. Volume 1254. (1997) 72–83
15. Colon, M., Uribe, T.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV. (1998) 293–304
16. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research (2000)
17. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 02: Programming Language Design and Implementation. (2002)
18. Barnett, M., DeLine, R., Fahndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3** (2004) 27–56
19. Detlefs, D., Nelson, G., , Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
20. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI 01: Programming Language Design and Implementation, ACM (2001) 203–213
21. Ball, T., Rajamani, S.K.: Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research (2002)
22. Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: TACAS 04: Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2004)
23. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN 00: SPIN Workshop. LNCS 1885. Springer-Verlag (2000) 113–130
24. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN 00: SPIN Workshop. LNCS 1885. Springer-Verlag (2000) 113–130
25. Ball, T., Rajamani, S.K.: Bebop: A path-sensitive interprocedural dataflow engine. In: PASTE 01: Workshop on Program Analysis for Software Tools and Engineering, ACM (2001) 97–103

26. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: Automatic theorem proving for predicate abstraction refinement. In: CAV 04: International Conference on Computer-Aided Verification. (2004)
27. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread modular abstraction refinement. In: CAV 03: International Conference on Computer-Aided Verification, Springer Verlag (2003) 262–274
28. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: CHARME 03: Advanced Research Working Conference on Correct Hardware Design and Verification Methods. (2003)
29. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: TACAS 01: Tools and Algorithms for the Construction and Analysis of Systems. (2001)
30. Owre, S., Shankar, N., Rushby, J.: PVS: A prototype verification system. In: CADE 11: International Conference on Automated Deduction. (1992) Saratoga Springs, NY.
31. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: CAV 04: International Conference on Computer-Aided Verification. (2004)
32. Filliatre, J.C., Owre, S., Rue, H., Shankar, N.: ICS: Integrated canonizer and solver. In: CAV 01: International Conference on Computer-Aided Verification. (2001)
33. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 355–367
34. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design **25** (2004) 105–127
35. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. (2001) 530–535
36. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In: Proceedings of CAV 2005. Volume 3576 of Lecture Notes in Computer Science., Springer Verlag (2005)
37. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Proceedings of LICS. (2002)
38. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2** (1986) 293–304
39. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV 02: International Conference on Computer-Aided Verification. (2002)
40. Strichman, O.: On solving presburger and linear arithmetic with SAT. In Aagaard, M., O'Leary, J.W., eds.: Formal Methods in Computer-Aided Design (FMCAD). Volume 2517 of Lecture Notes in Computer Science., Springer (2002) 160–170
41. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM (1992) 102–114
42. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In Hunt, W.A., Somenzi, F., eds.: Computer-Aided Verification (CAV). Number 2725 in LNCS, Springer-Verlag (2003) 141–153
43. Lahiri, S.K., Ball, T., Cook, B.: Predicate abstraction via symbolic decision procedures. In Etessami, K., Rajamani, S.K., eds.: Computer Aided Verification (CAV 2005). Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 24–38
44. Jain, H., Kroening, D., Sharygina, N., Clarke, E.: Word level predicate abstraction and refinement for verifying RTL Verilog. In: Proceedings of DAC 2005. (2005) 445–450
45. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. In: POPL, ACM Press (2004) 232–244