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 decision problems. A framework for reducing decision problems to propositional logic is described, which allows the surprising improvements in the performance of propositional SAT solvers to be exploited. The only assumption the framework makes is that an axiomatization of the desired logic is given.

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 their code.

More realistically, the goal is to prove or refute assertions given together with a 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 tremendously useful already.

The way these assertions are specified is intentionally left open; this may range from simplistic `assert()` statements inserted into the code to a formula 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, ranging from interactive theorem proving to automated methods such as Model Checking [2,3]. The workhorse of basically all software verification techniques is an efficient decision procedure, which decides validity of formulae in a suitably chosen assertion logic.

This paper discusses possible choices for this logic, and how to solve the resulting problems. We argue that any choice of assertion logic has to be sufficiently rich to permit expressions over the operators offered most commonly by the major programming languages. It also has to permit reasoning about dynamic data structures. Even for simplistic data structures, this requires support for quantification, either explicitly, in the form of universal and existential quantifiers, or implicitly, in the form of predicates that are defined by means of quantifiers.

* This research is supported by an award from IBM Research.

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 428–437, 2008.

A decision procedure for program analysis also needs to deal with decision problems that arise from refutation attempts, which are encodings of the feasibility of certain program paths. Given a large program, the paths can become very long, and consequently, the resulting formulae can become very large. The formulae often have a non-trivial propositional structure. Many decision procedures still perform case-splitting on the propositional structure, which limits the capacity of these tools severely.

Outline. In Section 2, we introduce the existing decision procedures that are used by program analysis tools, and discuss their suitability for this task. In Section 3, we propose a framework for encoding decision problems into propositional logic assuming an axiomatization of the assertion logic is given.

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 [4] and BOOGIE [5] use the Simplify [6] theorem prover to verify user-supplied invariants.

The SLAM [7,8,9,10,11,12] software model-checker uses ZAPATO [13] for symbolic simulation of C programs. The BLAST [14] and MAGIC [15] tools use Simplify for abstraction, simulation and refinement. Other examples include the Invest tool [16], which uses the PVS theorem prover [17]. Further decision procedures used in program verification are CVC-Lite [18], ICS [19], and Verifun [20].

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.

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

CBMC [21], a Bounded Model Checker for ANSI-C programs, uses a different approach: the program is unwound into a bit-vector logic formula, which is satisfiable if and only if there exists a trace up to a given length that refutes the property. This decision problem is reduced to propositional logic by means of circuit-encodings of the arithmetic operators. This allows supporting all operators as defined in the ANSI-C standard. The propositional formula is converted into CNF and passed to a propositional SAT solver. If the formula is satisfiable, a counterexample trace can be extracted from the satisfying assignment, which the SAT solver provides.

In [22], we proposed the use of such propositional SAT-solvers as a reasoning engine for automatic program abstraction. 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 [23] can now solve many instances with hundreds of thousands of variables and millions of clauses in a reasonable amount of time.

In [24], 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 Open Problems

The existing approaches are clearly not satisfying:

1. First of all, the word-level information about the variables is lost when splitting bit-vector operators 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.
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 [25]. 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 that solvers for QBF (quantified boolean formulae) make is encouraging, and promises to enable new applications just as the progress of SAT-solvers did.

A successful decision procedure for program analysis has to support a very rich logic, and be able to scale to large problem instances. In the next section, we discuss a framework for reducing decision problems to propositional logic assuming an axiomatization of the desired assertion logic is given.

3 Encoding Decision Problems

3.1 Propositional Encodings

Definition 1 (Propositional Encoding). *Let ϕ denote a formula that ranges over variables $v_1 \ldots, v_n$ from arbitrary domains $D_1, \ldots, D_n$. Let ϕ_P denote a propositional*

function ranging over the Boolean variables $b_1, \ldots, b_m$. The function ϕ_P is called a Propositional Encoding *of ϕ iff ϕ_P is equi-satisfiable with ϕ:*[1]

$$\begin{aligned} & \exists v_1, \ldots, v_n \in D_1 \times \ldots \times D_n . \phi(v_1, \ldots, v_n) \\ \iff & \exists b_1, \ldots, b_m \in \{0,1\}^m . \phi_P(b_1, \ldots, b_m) \end{aligned}$$

Once we have computed a propositional encoding of a given formula ϕ, we can decide satisfiability of ϕ by means of a propositional SAT solver. Linear-time algorithms for computing CNF for ϕ_P are well-known [26].

The first efficient proof-based reduction from integer and real valued linear arithmetic to propositional logic was introduced by Ofer Strichman [27]. The proof is generated using Fourier-Motzkin variable elimination for the reals and the Omega test for the integers [28]. We generalize the approach in [27] to permit arbitrary logics as long as a (possibly incomplete) axiomatization is provided.

Definition 2 (Propositional Skeleton)
Let $\mathcal{A}(\phi)$ denote the set of all atoms in a given formula ϕ that are not Boolean variables. The i-th distinct atom in ϕ is denoted by $\mathcal{A}_i(\phi)$. The Propositional Skeleton *ϕ_{sk} of a formula ϕ is obtained by replacing all atoms $a \in \mathcal{A}(\phi)$ by new Boolean variables $e_1, \ldots, e_\nu$, where $\nu = |\mathcal{A}(\phi)|$. We denote the identifier to replace atom $\mathcal{A}_i$ by $e(\mathcal{A}_i)$.*

As an example, the propositional skeleton of

$$\phi = (x = y) \wedge ((a \oplus b = c) \vee (x \neq y))$$

is $e_1 \wedge (e_2 \vee \neg e_1)$ and $\mathcal{A}(\phi)$ is $\{x = y, a \oplus b = c\}$.

Let E denote the set of variables $\{e_1, \ldots, e_\nu\}$, and let $\overline{e}$ denote the vector of the variables in E. Furthermore, let $\psi_a(p)$ denote the atom a with polarity $p \in \{\mathsf{true}, \mathsf{false}\}$:

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

Thus, $\psi_a(\mathsf{true})$ is the atom a itself, whereas $\psi_a(\mathsf{false})$ is the negation of a.

Lazy vs. Eager Encodings. Many decision procedures compute propositional encodings. All of them use the propositional skeleton as one conjunct of ϕ_P. The algorithms differ in how the non-propositional part is handled.

Let $x : \mathcal{A}(\phi) \longrightarrow \{\mathsf{true}, \mathsf{false}\}$ denote a truth assignment to the atoms in ϕ. Let $\Psi_{\mathcal{A}(\phi)}(x)$ denote the conjunction of the atoms $a_i \in \mathcal{A}(\phi)$ where a_i is in the polarity given by $x(a_i)$:

$$\Psi_{\mathcal{A}(\phi)}(x) := \bigwedge_{a \in \mathcal{A}(\phi)} \psi_a(x(a)) \tag{2}$$

Intuitively, $\Psi_{\mathcal{A}(\phi)}(x)$ is the constraint that must hold if the atoms have the truth values given by x. An *Eager Encoding* considers all possible truth assignments x before invoking the SAT solver, and computes a propositional encoding $\phi_E(x)$ such that

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

[1] Note that we do not require that the reduction is done in polynomial time, and thus, we can handle logics outside of NP.

The number of cases considered while building ϕ_E can often be dramatically reduced by exploiting the polarity information of the atoms, i.e., whether an atom 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_{sk}, and passed to a SAT solver. A prominent example of a decision procedure implemented using an eager encoding is UCLID [29].

A *Lazy Encoding* means that a series of increasingly stronger encodings $\phi_L^1, \phi_L^2, \ldots$ and so on with $\phi \Longrightarrow \phi_L^i$ is built. Most tools implementing a lazy encoding start off with $\phi_L^1 = \phi_{sk}$. 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, a truth assignment x^i to the atoms $\mathcal{A}(\phi)$.

The algorithm proceeds by checking if this assignment is consistent with the theory, i.e., if $\Psi_{\mathcal{A}\phi}(x^i)$ is satisfiable. If so, ϕ is satisfiable, and the algorithm terminates. If not so, a subset of the atoms $\mathcal{A}' \subseteq \mathcal{A}(\phi)$ that is already unsatisfiable under x^i is determined. The algorithm builds a *blocking clause* c, which prohibits this truth assignment to the atoms $\mathcal{A}'$. The next encoding ϕ_L^{i+1} is $\phi_L^i \wedge c$. Since the formula only becomes stronger, the algorithm can be tightly integrated into one run of a SAT-solver, which preserves the learning done by the solver in prior iterations. Advanced implementations of lazy encodings also preserve learning done within the decision procedure for the non-propositional theory.

Among others, CVC-Lite implements a lazy encoding of integer linear arithmetic. The decision problem for the conjunction $\Psi_{\mathcal{A}\phi}(x^i)$ is solved using the Omega test.

3.2 Propositional Encodings from Proofs

Proofs in any logic follow a pre-defined set of *proof rules*. A proof rule consists of a set of antecedents $A_1, \ldots, A_k$, which are the premises that have to hold for the rule to be applicable, and a consequence C. The rule is written as follows, where α denotes the "name" of the rule:

$$\frac{A_1, \ldots, A_k}{C}\,\alpha$$

A logic can be axiomatized by defining a set of special proof rules called axioms or axiom schemata, which define true statements in that logic. Many useful logics do not permit a complete axiomatization, but the set of axioms is usually sufficient to prove many theorems of practical interest.

Definition 3 (Proof Steps). *A* Proof Step s *is a triple* $(r, p, \mathcal{A})$*, where* r *is a proof rule,* p *a proposition (the consequence), and* $\mathcal{A}$ *a (possibly empty) list of antecedents* $A_1, \ldots, A_k$*.*

The fact that the dependence between the proof steps is directed and acyclic is captured by the following definition.

Definition 4 (Proof Graph). *A* Proof Graph *is a directed acyclic graph in which the nodes correspond to the steps, and there is an edge* (x, y) *if and only if* x *represents an antecedent of step* y*.*

Definition 5 (Proof-Step Encoder). *Let $\bot$ denote a contradiction, or the empty clause. Given a proof step $s = (r, p, \mathcal{A})$, its* Proof-Step Encoder *is a function $e(s)$ such that:*

$$e(s) = \begin{cases} \mathit{false} & : p = \bot \\ \neg e(p') & : p = \neg p' \\ \mathit{new\ propositional\ variable} & : \mathit{otherwise} \end{cases}$$

For a proof step $s = (r, p, \mathcal{A})$, we denote by $c(s)$ the constraint that the encoders of the antecedent steps imply the encoder of s, or more formally: if $\mathcal{A} = A_1, \ldots, A_k$ are the antecedents of s, then

$$c(s) := \left(\bigwedge_{i=1}^{k} e(A_i) \right) \longrightarrow e(p)$$

Definition 6 (Proof Constraint). *A* proof $P = \{s_1, \ldots, s_n\}$ *is a set of proof steps in which the antecedence relation is acyclic. The* Proof Constraint $c(P)$ *induced by P is the conjunction of the constraints induced by its steps:*

$$c(P) := \bigwedge_{s \in P} c(s)$$

A proof P is said to prove validity of ϕ if $e(\neg\phi) \wedge c(P)$ is unsatisfiable.

Theorem 1. *For any proof P and formula ϕ, ϕ implies $\phi_{sk} \wedge c(P)$.*

Thus, the idea of [27] 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. A proof P is obtained.
- Build ϕ_P as $\phi_{sk} \wedge c(P)$.
- The prover must be modified to obtain *all* possible proofs, i.e., must not terminate even if the empty clause is resolved.

Example. We illustrate the algorithm above with the following Hoare triple:

$$\{b = 5 \wedge (p \mapsto a \vee p \mapsto b)\}\ *p := 3\ \{b = 5\}$$

Informally, let '$p \mapsto a$' denote the fact that a pointer p points to some variable a. We denote the dereferencing operation of a pointer p by $*p$. Let $c?x : y$ denote x if c holds, and y otherwise. Given a suitable definition of assignment to $*p$, the following verification condition ψ could be generated for the triple above:

$$\begin{aligned} \psi := \quad & (b' = 5 \wedge (p \mapsto a \vee p \mapsto b) \wedge \\ & (p \mapsto a ? a = 3 : a = a') \wedge \\ & (p \mapsto b ? b = 3 : b = b')) \\ \longrightarrow\ & b = 5 \end{aligned}$$

As we aim at showing validity of ψ, we form $\phi := \neg\psi$, and check satisfiability of ϕ. A propositional encoding of ϕ can be obtained using the following mapping from atoms to variables:

$$\begin{array}{ll} e(b' = 5) = v_0 & e(a = a') = v_4 \\ e(p \mapsto a) = v_1 & e(b = 3) = v_5 \\ e(p \mapsto b) = v_2 & e(b = b') = v_6 \\ e(a = 3) = v_3 & e(b = 5) = v_7 \end{array}$$

Consequently, the propositional skeleton ϕ_{sk} is:

$$\begin{array}{l} v_0 \wedge (v_1 \vee v_2) \wedge \\ (v_1 \longrightarrow v_3) \wedge (\neg v_1 \longrightarrow v_4) \wedge \\ (v_2 \longrightarrow v_5) \wedge (\neg v_2 \longrightarrow v_6) \wedge \\ \neg v_7 \end{array}$$

Reasoning for equality logic is sufficient to prove or disprove claims of the form of our example. The only proof rule needed is transitivity of equality[2]:

$$\frac{a = b, b = c}{a = c}$$

An instance of this rule is $(b = 3 \wedge b = b') \longrightarrow b = 5$, which yields the constraint

$$(v_0 \wedge v_6) \longrightarrow v_7$$

Let this constraint conjoined with ϕ_{sk} be denoted by ϕ_{enc}. The formula ϕ_{enc} can be passed to a SAT solver. One of the satisfying assignments that the SAT solver could produce is v_0,$\neg v_1$,v_2,$\neg v_3$,v_4,v_5,$\neg v_6$,$\neg v_7$, i.e., $p \mapsto b$, and $b = 3$, which refutes the claimed post-condition.

3.3 Proofs for Program Verification

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

Even if great care is taken to obtain a small set of axioms, the number of proofs is still too large. Furthermore, in the case of bit-vector logic, the proofs will include derivations that are based on reasoning about single bits of the vectors involved, resulting in a flattening of the formula, which resembles the circuit-based models used for encodings of bit-vector logic into propositional logic.

We therefore propose to sacrifice precision in order to be able to reason about bit-vectors, and compute an over-approximation of ϕ_P. This does not necessarily imply

[2] We also use the fact that equality is symmetric and axioms about integers; however, these facts are usually hard-coded into the procedure that applies the transitivity rule.

that the program analysis tool will become unsound. In fact, most existing program analysis tools, e.g., SLAM and BLAST, use decision procedures that compute over-approximations in order to save computational effort. Such over-approximations can be refined automatically if needed, e.g., based on UNSAT cores as in [30] or based on interpolants as in [31].

One trivial way to obtain an inexpensive over-approximation of ϕ_P is for example, bounding the depth of the proofs. Future research could, for example, focus on better proof-guiding heuristics.

The technique described above is applicable to decision problems, e.g., for checking verification conditions, and to quantification problems, as arising in fixed-point computations. For an explanation how this technique can be applied to quantification problems arising in predicate abstraction, we refer the reader to [32,33].

4 Conclusion

Program verification engines rely on decision procedures. However, despite 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. A possible logic to model the pointer operations is separation logic.

The current state-of-the-art for deciding bit-vector logic is an ad-hoc approach using propositional SAT-solvers. An efficient decision procedure that supports 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, 63–69 (2003)
2. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
3. Clarke, E.M., Emerson, E.A.: Synthesis of synchronization skeletons for branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
4. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002: Programming Language Design and Implementation, pp. 234–245 (2002)
5. Barnett, M., DeLine, R., Fahndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3, 27–56 (2004)
6. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
7. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI 2001: Programming Language Design and Implementation, pp. 203–213. ACM, New York (2001)
8. Ball, T., Rajamani, S.K.: Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research (2002)

9. Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 388–403. Springer, Heidelberg (2004)
10. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
11. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
12. Ball, T., Rajamani, S.K.: Bebop: A path-sensitive interprocedural dataflow engine. In: PASTE 2001: Workshop on Program Analysis for Software Tools and Engineering, pp. 97–103. ACM, New York (2001)
13. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: Automatic theorem proving for predicate abstraction refinement. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 457–461. Springer, Heidelberg (2004)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread modular abstraction refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
15. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 19–34. Springer, Heidelberg (2003)
16. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 98–112. Springer, Heidelberg (2001)
17. Owre, S., Shankar, N., Rushby, J.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
18. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)
19. Filliatre, J.C., Owre, S., Rue, H., Shankar, N.: ICS: Integrated canonizer and solver. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, Springer, Heidelberg (2001)
20. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 355–367. Springer, Heidelberg (2003)
21. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of DAC 2003, pp. 368–371. ACM Press, New York (2003)
22. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design 25, 105–127 (2004)
23. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC, pp. 530–535. ACM, New York (2001)
24. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 296–300. Springer, Heidelberg (2005)
25. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Proceedings of LICS, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
26. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. 2, 293–304 (1986)
27. Strichman, O.: On solving Presburger and linear arithmetic with SAT. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 160–170. Springer, Heidelberg (2002)

28. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM, 102–114 (1992)
29. 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: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 78–92. Springer, Heidelberg (2002)
30. Jain, H., Kroening, D., Sharygina, N., Clarke, E.: Word level predicate abstraction and refinement for verifying RTL Verilog. In: Proceedings of DAC, vol. 2005, pp. 445–450 (2005)
31. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. In: POPL, pp. 232–244. ACM Press, New York (2004)
32. Lahiri, S.K., Ball, T., Cook, B.: Predicate abstraction via symbolic decision procedures. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 24–38. Springer, Heidelberg (2005)
33. Kroening, D., Sharygina, N.: Approximating predicate images for bit-vector logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 242–256. Springer, Heidelberg (2006)