Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Approximating the Safely Reusable Set of Learned Facts ⋆

Domagoj Babić, Alan J. Hu

Computer Science Department
University of British Columbia

The date of receipt and acceptance will be inserted by the editor

Abstract. Despite many advances, today's software model checkers and extended static checkers still do not scale well to large code bases, when verifying properties that depend on complex interprocedural flow of data. An obvious approach to improve performance is to exploit software structure. Although a tremendous amount of work has been done on exploiting structure at various levels of granularity, the fine-grained shared structure among multiple verification conditions has been largely ignored. In this paper, we formalize the notion of shared structure among verification conditions, and propose a novel and efficient approach to exploit this sharing by safely reusing facts learned while checking one verification condition to help solve the others. Experimental results show that this approach can improve the performance of verification, even on path- and context-sensitive and dataflow-intensive properties.

1 Introduction

Recent advances in formal verification have brought the long-time dream of automatic formal verification of software closer to reality. The hope is that a programmer would need only specify desired correctness properties — or the verification tool could have pre-specified properties, such as proper locking-unlocking, or adherence to an Application Programming Interface — and the tool would fully automatically construct the formal logical model and verify whether the desired correctness properties hold.

Verification conditions (VCs) are the logical formulas, constructed from a system and desired correctness properties, such that the validity of the verification conditions corresponds to the correctness properties holding. Constructing and proving VCs are both essential steps in software verification, and both have been active areas of research. In this paper, we focus on proving the validity of VCs more efficiently.

The trend today is to use automated decision procedures to prove or disprove the computed VCs. Unfortunately, this process is computationally extremely expensive and is the main bottleneck to the wider application of formal and semi-formal software verification methods. Previous work has focused on the computation of VCs (e.g., [1,2]), abstraction to make the VCs simpler for the decision procedure (e.g., [3, 4]), and the efficiency of the decision procedures themselves (e.g., [5–9]).

This paper explores a different direction for improving efficiency — namely, exploiting shared structure among multiple VCs at the level of individual expressions — and proposes a technique that exploits this structure. Since solving VCs is typically expensive, elimination of this redundancy has the potential to significantly improve performance of static checking. In this paper, we present our insights, formalize the notion of shared structure, and propose an algorithm for exploiting this shared structure by safely reusing facts learned while checking one verification condition to help solve the others. We provide experimental evidence that our approach can cut runtime by almost one third and reduce the number of timeouts.

2 Background

The work in this paper fits in the context of static checking of software. The distinction between static checking (e.g., [10–13]) and model checking [14,15] has become fuzzy, but historically, static checking has emphasized fast bug hunting and scalability to large software, at the expense of precision (and often soundness and/or completeness), whereas model checking has emphasized precision and soundness, with the primary research challenge being scalability. In both paradigms, though, the verification task typically consists of checking a

⋆ This work was supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada and a Microsoft Research Graduate Fellowship. This paper is based on and extended from a paper published in the 2007 Haifa Verification Conference.

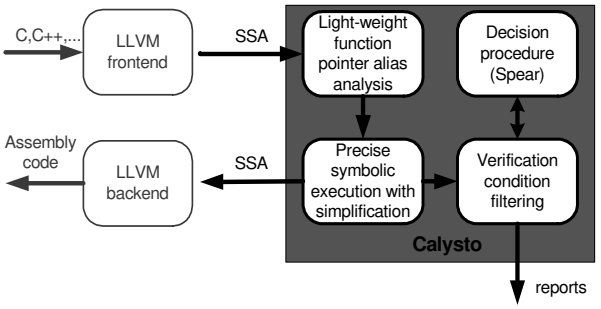

Fig. 1. High-Level CALYSTO Architecture

large number of properties, e.g., there could be thousands of user-supplied assertions in a large codebase, or even more automatically generated checks, such as that pointer dereferences are non-null, array accesses are in-range, locking protocols are observed, etc. Our goal in this paper is to exploit the shared structure among the many verification conditions.

More precisely, we have noticed that many VC share common formulas, which usually correspond to prefixes/suffixes of shared paths in the analyzed program. One way to exploit this insight is to use classical incremental satisfiability [16–18]. If constraints of one VC were a subset of constraints of the next one, we could simply take all the constraints (clauses, in the context of SAT solving) learned while solving the first VC and reuse them when we move to the next VC. Unfortunately, in practice, VCs usually share only a subset of constraints and it is rare to find pairs of VCs that can be ordered by the subset relation. So, when moving from one VC to another, decision procedures have to remove the constraints that do not belong to the next VC being solved, as well as all the learned constraints dependent on the removed set of constraints. This can be done, but requires tedious bookkeeping of all dependencies among the original and learned constraints, i.e., one has to record a potentially very large implication graph. In this paper, we present a very lightweight technique for approximating the set of constraints that can be safely reused, while avoiding the need for keeping the large implication graphs.

Our work is based on our extended static checker CALYSTO [19]. The high-level architecture of CALYSTO is shown in Fig. 1. CALYSTO is designed as a compiler pass, in the spirit of Hoare's "verifying compiler" grand challenge [20]: it accepts the compiler's intermediate representation, in static single assignment (SSA) form [21], performs various verification checks, issues bug reports and warnings, and then passes semantically unmodified SSA on to the compiler backend.

Internally, the CALYSTO system consists of three stages, supported by an automatic theorem prover, SPEAR. The first stage is a lightweight function pointer alias analysis, which constructs (a sound approximation of) the call graph, including indirect calls through function pointers. We sacrifice some precision, by using a sound, flow-sensitive, but context-insensitive alias analysis, tracking only the function pointers. The next stage is symbolic execution [22], which executes the program using symbolic instead of concrete values. CALYSTO symbolically executes functions in the analyzed program, computing symbolic definitions for each modified variable and memory location. These symbolic definitions are used to create the verification conditions (VCs). The symbolic execution machinery allows generating VCs for any assertion at any point in the program. CALYSTO currently supports user-supplied assertions (written as boolean expressions in whatever programming language the compiler front-end is parsing), but in the spirit of static checking, also automatically generates VCs to check that each pointer dereference cannot be NULL. The last stage consists of checking and filtering the verification conditions. In principle, the VCs could be sent directly to a theorem prover for checking, and this approach is used in most other tools. We have found, however, that both efficiency and usability can be improved by control and filtering of what VCs are checked. The main efficiency gain is due to *structural abstraction* [23], where CALYSTO exploits natural function-call boundaries for abstraction and lazily refines only those function effects that are needed to refute or demonstrate a concrete violation of a VC. The actual validity-checking of VCs is done by SPEAR, which is a sound and complete, fully automatic theorem prover that supports Boolean logic, bit-vector operations, and bit-accurate arithmetic.

Although the work in this paper is built on CALYSTO, we believe that the contribution of this paper can be applied to any static checker or software model checker that uses a decision procedure to check VCs, assuming some reasonable properties of the VCs (see Sec. 3). Considerable research went into the specific design decisions behind CALYSTO's architecture [19,24], but the general pattern — a compiler front-end to analyze the software, a formal analysis to generate multiple VCs for multiple program properties, and a decision procedure back-end to check the VCs – is common.

3 Definitions and Assumptions

In this section, we give definitions of some basic concepts required for understanding the rest of the paper and present the assumptions on which our method relies.

Verification Conditions (VCs). VCs are logical formulas, constructed from a system and desired correctness properties, such that the validity of VCs corresponds to the correctness of the system. Commonly, correctness properties in programs are specified with assertions, which can be either written by programmers, or automatically generated (e.g., [25]). If all the assertions in the program are valid, the program is considered to be correct with respect to the given set of assertions.

A predicate that represents the condition under which some assertion is reachable in the program code will be called the reachability predicate. Let R stand for a reachability predicate and α be the asserted condition. The VC is then given by $R \Rightarrow \alpha$, intuitively saying that if there is a feasible path to α, the assertion condition should be valid.

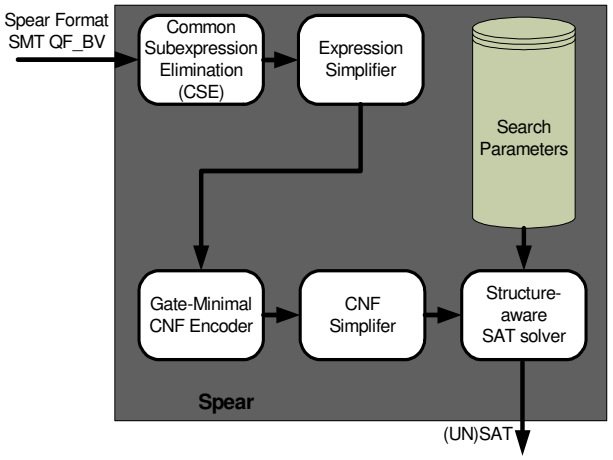

Fig. 2. High-Level SPEAR Architecture

Verification conditions can be computed by intra- or inter-procedural analysis. To achieve high precision, approaches based on intra-procedural analysis require user-provided interface invariants. In practice, however, programmers seldom provide these invariants. Verification conditions computed by inter-procedural analysis tend to be large and based on code spread throughout many function calls.

Decision Procedure. We are interested in bit-precise software verification in order to be able to catch frequent integer under/over-flow bugs[1]. So, all of our analysis will be assuming bit-vector arithmetic. Our bit-vector arithmetic decision procedure SPEAR[2] is based on a SAT solver and supports all standard bit-vector arithmetic operators on finite bit-vectors, including expensive operators (like multiplication and division). Although we use bit-vector arithmetic, the contribution is largely independent of the chosen logic.

SPEAR's architecture is illustrated in Fig. 2. SPEAR simplifies expressions in two steps: First, common subexpression elimination merges (structurally) equivalent expressions. Common subexpression elimination (CSE) is a simple, but very important optimization for decision procedures based on SAT solvers. SPEAR eliminates common subexpressions by simple structural hashing. Second, a simple term-rewriting engine simplifies the expressions starting at the leaves of the maximally-shared graph representing the formula and moving toward the root. The engine performs operations like constant-propagation (e.g., $a+0=a$), constant-collection (e.g., $a+1+2=a+3$), simple deduction (e.g., $a<b \wedge a>b = \mathsf{false}$), redundancy elimination (e.g., $ITE(\phi,a,a)=a$), partial canonicalization (e.g., $ite(\phi,\phi\wedge a,b)=ite(\phi,a,b)$), strength-reduction (e.g., $3*a=a<<1+a$), and so on.

Programs contain non-linear operators, and to be bit-precise, one must have a decision procedure that supports them. A number of different methods have been developed for linear bit-vector arithmetic, but few of them are applicable to non-linear operators. We use the bit-blasting approach: Variables are encoded as bit-vectors of suitable size, and operators are replaced by digital circuits corresponding to that operator. In effect, VCs become large digital circuits, which can be converted to CNF using Tseitin's transform [27] and given to a SAT solver. We use gate-optimal circuit encoding, trying to minimize the number of gates.

The CNF simplifier performs variable elimination [28] and elimination of satisfied clauses and falsified literals [29].

The core of SPEAR is a DPLL-style [30] SAT solver. SPEAR incorporates a number of novel optimizations. However, one of the most important features of SPEAR and its core is configurability: every single search parameter, which is typically hard-coded in most off-the-shelf decision procedures, can be set on the command line. The full flexibility and power of this configurability becomes obvious in combination with automatic tuning [31].

When automated decision procedures are used for proving VCs, the validity of a verification condition *VC* is usually proven by asking the decision procedure to prove unsatisfiability of the formula $VC = \mathsf{false}$. Its satisfiability means that there is a possible bug in the program from which the VC was constructed.

In our setting, we wish to check multiple VCs. Therefore, to identify and use the sharing between the VCs, we cache subterms of the VCs (and clean the cache periodically), while using classical structural hashing to reuse the expressions already present in the cache. When we learn a context-independent fact about a certain term (using the techniques presented in this paper), we associate it with the term itself. Later, if the term becomes a part of another VC, we automatically add the context-independent fact when we pass the new VC to the decision procedure.

Representation. As mentioned, we represent VCs as acyclic graphs. This representation simplifies the reasoning about the structure of the formulas. In addition, using simple node hash tables, we eliminate all common sub-expressions. Such graphs, in which all redundancies have been eliminated, are known as maximally-shared graphs:

Definition 1 (Maximally-Shared Graph).
Given an acyclic graph $G=(N,E)$, let $\mathscr{L}$ stand for a labeling function $\mathscr{L}: N \longrightarrow string$. Define the arity of a node n, denoted as $|n|$, as the number of outgoing edges. The outgoing edges are ordered, and the i-th edge of a node n will be denoted as $child_i(n)$. Two operator nodes n_1 and n_2 are defined to be equivalent ($n_1 \triangleq n_2$) if and only if $|n_1|=|n_2|$, $\mathscr{L}(n_1)=\mathscr{L}(n_2)$, and

$$\forall i: 0 \leq i \leq |n_1| : child_i(n_1) \triangleq child_i(n_2).$$

(This is standard bisimulation equivalence, but applied to a graph representing the static structure of a VC, rather than the more typical application to a transition system.) Graph G is maximally-shared if $\neg\exists n_1,n_2 \in N : n_1 \neq n_2 \wedge n_1 \triangleq n_2$.

CALYSTO computes verification conditions directly as maximally-shared graphs. The graph representation can be transformed into a conjunction of expressions by standard renaming. We shall identify nodes in the graph with the variables

[1] For instance, the 2004 JPEG security exploit (see e.g., [26]).

[2] http://www.domagoj.info/index_spear.htm

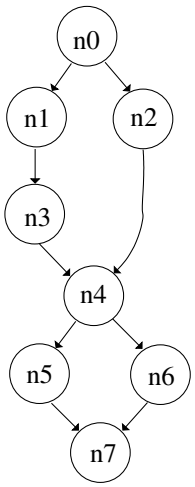

Fig. 3. Dominance Relation. Given the acyclic graph above, some dominance relationships include: $n_1 \geqslant n_1, n_1 \geqslant n_3, n_0 \gg n_3, n_0 \gg n_5, n_4 \gg n_6$, and $n_2 \not\gg n_4$.

used for renaming. This is a one-to-one mapping. We shall represent equality (resp. disequality) in formulas and algorithms as $=$ (resp. $\neq$), while in the code snippets and graphs = will stand for assignment, and == (resp. !=) for equality (resp. disequality).

Graph Relations. If there is an edge connecting two nodes, $n \longrightarrow m \in E$, then n is a *predecessor* of m, and m is a *successor* of n. The set of predecessors of a node n will be denoted as $Pred(n)$, and the set of its successors as $Succ(n)$. The nodes in the transitive closure of $Pred(n)$ are *ancestors* of n, and the nodes in the transitive closure of $Succ(n)$ are *descendants* of n, denoted $Desc(n)$.

Dominance Relation. To analyze the shared subgraphs, we rely upon the dominance relation [32]:

Definition 2 (Dominance Relation).
A node n dominates node m if and only if all the paths from the entry node to m go through n, written as $n \geqslant m$. If $n \neq m$, n strictly dominates m, denoted $n \gg m$.

The dominance relation (illustrated in Fig. 3) is a partial order (reflexive, antisymmetric, and transitive) and can be computed in $\mathcal{O}(N\alpha(E,N))$ [33] time, where α is the extremely slowly growing inverse of Ackermann's function. In practice, a simpler $\mathcal{O}(E \log N)$ algorithm [33] is faster, even for very large graphs, and that is what we are using for the results in this paper.

The dominance relation, as defined above, requires a unique entry node. The technique presented in this paper always considers the root node that represents a single VC to be the entry node for the computation of the dominance relation.

Assumptions. The work presented in the paper relies on several assumptions, which are either almost always satisfied in practice or can be satisfied with a trivial amount of post-processing.

First, the decision procedure must be able to identify facts of the form *variable* = *constant* that are implied by formulas being solved. For instance, if the decision procedure is based on a SAT solver, learned unit literals are such facts. Decision procedures based on the Nelson-Oppen [9] framework generate conjunctions of equalities (providing that the individual theories are convex), and it is easy to extract the equalities that satisfy our requirement.

Second, we assume complete propagation of equalities with constants, i.e., we require that the decision procedure generates facts of the form $a = 7, b = 7, c = 7$ instead of $a = 7, b = a, c = b$. This is trivial to accomplish by a linear time constant propagation post-processing even if the decision procedure does not make such guarantees. Assuming that the formula is satisfiable, both SAT solvers and E-graphs [34], on which the Nelson-Oppen framework is based, satisfy this requirement.

Third, we assume that the proper sub-expressions of a VC are logically consistent. Every expression that can be translated into an acyclic circuit-like representation satisfies this requirements because circuits themselves are logically consistent — every input produces some output. If the consistency assumption were violated, then the decision procedure could imply arbitrary implicants. The consistency assumption ensures that the implicants derived from a sub-expression are meaningful. Two small examples provide the intuition behind this assumption.

Example 1. Consider an obviously inconsistent formula $a < 0 \wedge a > 0$. By introduction of fresh variables $n_0, \cdots, n_2$ we get:

$$n_0 = a < 0$$
$$n_1 = a > 0$$
$$n_2 = n_0 \wedge n_1$$

This is a logically consistent set of constraints which corresponds to the circuit-like representation in Fig. 4. Note that the constraints force n_2 to be always false, but the constraints themselves are satisfiable. Variable n_2 corresponding to the root node in Fig. 4 can be seen as a *circuit output*. □

As mentioned earlier, the goal is to prove validity of a VC, i.e., that the value of the output node is always true. We can check this by adding the constraint *root_node* = false and then checking satisfiability. If the resulting formula is satisfiable, the original VC is not valid. Only by adding the additional constraint can the constraints become inconsistent, as in the next example.

Example 2. Given the formula: $VC = (a > b \Rightarrow a \geq b)$, we can construct the set of constraints:

$$n_0 = a > b$$
$$n_1 = a \geq b$$
$$n_2 = n_0 \Rightarrow n_1$$

which is consistent. Now, to check validity, we add constraint n_2 = false to the set, forcing the output to false. The set of

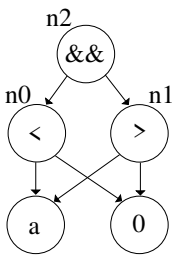

Fig. 4. Small Maximally-Shared Graph Representing $a < 0 \wedge a > 0$. Successors of non-commutative operators are ordered in the natural order (from left to right). Operator nodes are labelled with the operator (inscribed) and the name of corresponding variable used in renaming (adjacent to the node).

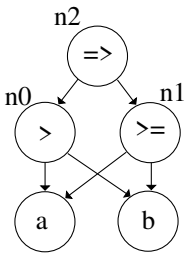

Fig. 5. Graph Corresponding to the Set of Constraints in Example 2.

constraints becomes unsatisfiable, meaning that the original VC was valid. □

4 Exploiting Shared Structure

In software, many paths share common statements, which means that computed VCs will share common sub-expressions. However, it is less obvious how to exploit that structure.

A direct approach would be to construct a single formula as the disjunction of all (negated) verification conditions, give it to the theorem prover, and for each solution, report a bug, then add a blocking clause to eliminate the failing verification condition (a single disjunct) from further consideration.[3] Everything that the theorem prover learns while checking one (disjunct in the) VC can be re-used when looking for additional solutions, so this is a "perfect solution". Unfortunately, it suffers from the same problem as clause learning in a SAT solver: there is too much information that is learned, with very little of it being useful later. The information overload is especially problematic because our verification analysis is interprocedural: the brute-force direct approach would compute the single VC (the disjunction of the negated VCs for all assertions) for the entire program, potentially forcing the decision procedure to explore completely different parts of the verified program in a single run. The added blocking clauses are bound to become obsolete at certain point, but it is hard to detect that without having any information about the structure of the program. Alternatively, heuristic removal of blocking clauses might produce repeated error traces corresponding to the same assertion violation, or even worse, failure to terminate.

Instead, in CALYSTO, the approach we use is to compute a single VC per assertion in each context. Given a predicate ϕ representing a VC in the caller, we *lift* the VC into the calling context in two steps: First, we substitute the formal parameters of the callee with the actual parameters in the caller's context (the same substitution is carried out for the memory locations that the callee accesses). Second, the predicate that represents the conditions under which the callee is called, say ψ, is used to construct an implication $\psi \Rightarrow \phi$, which represents our VC lifted to the caller's context.[4]

Computing a single VC per assertion in each context slices out the part of the program that is relevant to the assertion and context we are interested in, drastically improving the performance of decision procedures. Once a VC is either proved valid, or a counterexample is found, the entire VC is discarded, and we proceed to the next VC. While such an approach avoids bloating the clause database with blocking clauses, it also, unfortunately, discards the knowledge gained about sub-expressions shared by multiple VCs.

In this paper, we seek to distill out implicants learned while solving one VC that are useful for solving another VC. Regrettably, not all implicants can be re-used, because they can depend on the context of the first VC, which might not be true of the other VC. The crux of the problem is that decision procedures can propagate information in any direction. Consider the VC shown in Fig. 5 with the additional constraint $n_2 = \mathsf{false}$. Most decision procedures would start solving the VC by propagating constants. From $n_2 = \mathsf{false}$, it follows that $n_0 = \mathsf{true}$ and $n_1 = \mathsf{false}$. From $n_1 = \mathsf{false}$ it follows that $a < b$. The last implicant contradicts $a > b$, hence the set

[3] In a simple usage model of formal verification, a tool might naturally stop as soon as it finds a false VC and report an error. In the static-checking style that we follow, however, where there may be many thousands of automatically generated assertions to check, and where there may be occasional false error reports, it is much more useful to try to recover from errors and continue checking as many assertions as possible on each run, much as a compiler will report multiple errors during a compilation. Hence, we need a mechanism to continue checking VCs despite finding a failing VC.

[4] A naïve application of this technique leads to an exponential blowup, as more and more VCs have to be lifted through the call graph. An optimization technique that we implemented in CALYSTO lifts only VCs that depend on formal parameters, globals, or memory locations reachable through formals or globals. While this technique can lead to spurious errors being reported from unreachable code, we found that in practice a simple dead-code elimination pass eliminates a vast majority of such spurious warnings. An even better approach would be to check the validity of each VC before lifting: valid VCs are independent of the calling context and need not be lifted, and counterexamples that are independent of the calling context can be reported as local bugs right away. Although we believe that such an optimization would be extremely valuable, we haven't had time to implement it. A more detailed discussion is given in [24].

of constraints represented by the graph is unsatisfiable. This propagation of information from *above* introduces assumptions that might not hold in all other contexts. Any other VC that contains the sub-expression represented by n_2 and does not enforce $n_2 = \mathsf{false}$ cannot reuse the previously computed solution.

Intuitively, we want a way to figure out which implicants were implied from *below*. For instance, if a decision procedure can infer that node n_2 is always true just by considering its descendants, then the same decision procedure will be able to infer the same result if n_2 appears as a sub-expression of any other VC. In other words, $n_2 = \mathsf{true}$ becomes a *context-independent invariant*.

The concept of "context" can be defined in many ways. Since we study the fine-grained structure of expressions computed from software, it is helpful to define context on the maximally-shared graphs as follows: We say that an expression represented by a node in a maximally-shared graph is context-independent if its value is uniquely implied by its sub-expressions, otherwise the relation is context-dependent. For instance, in Example 2 (Fig. 5) the implicant $n_0 = \mathsf{true}$ is context-dependent because the implication chain came from the predecessor n_2. On the other hand, $n_2 = \mathsf{true}$ is a context-independent invariant as it follows from the nodes below n_2.

Decision procedures can generate a large number of implicants. For example, SAT solvers usually generate a single implicant per conflict. Keeping even only 10% of implicants from each VC requires excessive amounts of memory. In addition, not all implicants are context-independent invariants. So, we use a more restricted form of invariants to represent learned facts:

Definition 3 (Node Fixation). Let n be some node in a maximally-shared graph and ψ an invariant derived by the decision procedure of the form $n = constant$. We shall say that n is *fixed by the decision procedure*. Define predicate $fix_{DP}(n)$ to be true iff n is fixed by the decision procedure. If $fix_{DP}(n) = \mathsf{true}$, define operator $FixVal_{DP}(n)$ to be an operator that returns the *constant* to which the node n was fixed.

The invariants derived by the decision procedure represent knowledge gained about the solved VC; these invariants can be either context-dependent or context-independent. We need to separate out the context-independent ones, as those can be used later when other VCs are solved. So, we define a subset of nodes that were fixed by the decision procedure in a context-independent manner as:

Definition 4 (Fixation from Below). Let n be a node fixed by the decision procedure to $FixVal_{DP}(n)$. If the invariant $n = FixVal_{DP}(n)$ was derived only by considering a subgraph rooted at n, we shall say that n was *fixed from below*. Define predicate $fix_{\uparrow}(n)$ to be true iff n is fixed from below.

There are two basic approaches to establishing context independence. First, the decision procedure could record the implication graph for each inferred relation. Second, one could attempt to reconstruct the chain of reasoning from the relations produced by the decision procedure once it terminates. In our experience, the first approach is impractical for decision procedures based on SAT solvers, as it requires excessive resources, and slows down the core of the solver by several orders of magnitude. However, it might be a viable approach within the Nelson-Oppen framework if all the combined theories are convex [9][5]. We present a reconstruction-based approach: a simple algorithm that given a set of nodes fixed by the decision procedure, efficiently computes a safe approximation of the set of nodes fixed in a context-independent manner.

4.1 Algorithm

Depending on the client, the queries to the decision procedures might be available all at once, or computed in a lazy manner. For example, a static checker that relies on some form of abstraction might compute incrementally more refined VCs, or process the call graph of the verified application in an incremental manner. Other clients, like invariant generators, might construct a number of queries at once, and ask for invariants common to all the queries. Because CALYSTO performs structural abstraction [23], we focus on the case where queries are posed in an online manner: VCs are checked one-by-one and future queries are not known. Obviously, the same algorithm can also handle the case where all VCs are available in advance.

Algorithm 1 computes a safe approximation of the set of nodes that are fixed from below. The values of nodes fixed from below are stored in an associative table *Fixed*, indexed by the nodes. Later, if another VC contains a node n that exists in the table, the value that is read from the table, $Fixed[n]$, is used to create an additional constraint $n = Fixed[n]$. Adding this additional constraint to the set of constraints representing the VC being solved saves computation effort because the decision procedure can immediately start propagating the $Fixed[n]$ constant.

Line 4 performs some basic technical checks. The value of the root node is fixed from above (to false because we are checking for unsatisfiability), so the root node is eliminated from consideration. Note that there is no reason why the root node couldn't be fixed from below as well. However, in that case, our analysis is not capable to resolve whether the implication chain came from above or from below. In order to resolve this ambiguity, the theorem prover would need to track implication graphs — a technique which we consider too expensive.

Only three basic types of nodes can be present in the expression graph: constants, variables, and operators. Constants are always fixed from below, variables are always considered unconstrained, so it makes sense to attempt to fix the values of only the operator nodes.

[5] Modular arithmetic, as well as the theory of integers, are not convex, so even decision procedures based on Nelson-Oppen framework would need some form of bookkeeping, similar to implication graphs, to be able to exactly identify a set of assumptions from which each implicant was implied.

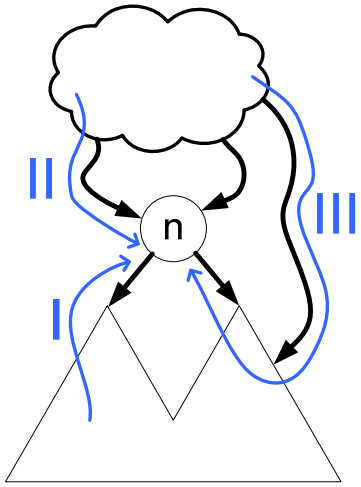

Fig. 6. Context-Independent (I) and Context-Dependent (II,III) Propagation of Knowledge.

Algorithm 1 Approximation of the Set of Nodes Fixed from Below. Predicate $isConstant(n)$ returns true if the node n is a constant node, predicate $isRoot(n)$ returns true if the node n represents a VC (root of the graph), while $isOperator(n)$ is true iff n represents an operator. Results of the analysis are stored in the table *Fixed*, indexed by nodes. The set of descendants (resp. predecessors) of a node n is denoted as $Desc(n)$ (resp. $Pred(n)$).

```
1:  procedure FIX(n, Fixed)
2:      for each s ∈ Succ(n) do
3:          FIX(s, Fixed)
4:      if ¬isRoot(n) ∧ isOperator(n) ∧ fix_DP(n) then
5:          for each d ∈ Desc(n) do
6:              if ¬isConstant(d) ∨ n ⋙ d then
7:                  return
8:          for each p ∈ Pred(n) do
9:              if fix_DP(p) then
10:                 return
11:         Fixed[n] ← FixVal_DP(n)
```

Intuitively, the algorithm works as follows. Lines 5–7 check whether the node dominates all its descendants. If n does not dominate some descendant d, it follows that d is reachable from the root of the graph by at least one path that does not go through n. Consequently, d appears in at least two contexts (one represented by the path that passes through n and the other by path that avoids n). Without reconstructing the implication graph that led the decision procedure to imply $n = FixVal_{DP}(n)$, it is not possible to distinguish between these cases: (I) The invariant was implied from below, relying only on the descendants of n. (II) The invariant was implied from above, possibly all the way from the root node. (III) The constant propagation chain came from above, avoiding n, fixed the value of some descendant of n, which in turn implied the invariant. All three cases are illustrated in Fig. 6. The dominance test eliminates the third case. The purpose of lines 8–10 is to eliminate the second case. Obviously, if no predecessor of n was fixed, the constant propagation chain must have come from below. Remember that we assume complete propagation of constants, so each constant propagation chain has to have its beginning and its end. The nodes that pass both tests can be safely considered fixed from below.

Implementations should mark visited nodes and avoid revisiting them. As each node has to be visited only once, and each node can have at most $|N|$ descendants and predecessors together (G is acyclic), the worst case complexity is $\mathcal{O}(|N|^2)$, but that is a very pessimistic bound, especially since our VCs tend to be sparse graphs in practice. We found that in practice the algorithm runs almost in linear time if a depth-first-search is used to iterate over the descendants in lines 5–7. Intuitively, the deeper the node is, the larger the probability that it is shared (simpler expressions are more frequently shared than complex ones). Hence, the probability of running into a node not dominated by n is becoming larger as we get further away from n (downwards). The dominance relation can be computed in $\mathcal{O}(|N|\alpha(|N|,|E|))$, as noted before.

How good is the approximation? The algorithm is able to fix only the nodes that are at the end of a constant propagation chain. Intuitively, the last fixed node in the constant propagation chain is the node that required the largest amount of reasoning. For instance, let $n_1, \cdots, n_k$ be a sequence of nodes whose values were fixed from below, all lying on the same path. Assume that there are k VCs such that the first contains n_1, the second n_2 but not n_1, and so on. The last VC contains only n_k. Since all node values were fixed from below, it is likely that the decision procedure will repeat the same steps while solving each of those k VCs, so eventually, all nodes in the constant propagation chain might become fixed from below, and constraints $n_i = FixVal_{DP}(n_i)$ can be used later if any of the n_i nodes becomes a part of other VCs. Even though this approximation is crude, it is very fast even for large VCs. In Sec. 5, we will evaluate whether the algorithm is fast enough and can find enough context-independent invariants to improve overall performance.

To prove that Alg. 1 really computes a set of nodes fixed from below, we start with the following lemma.

Lemma 1. *Let n be the subgraph of graph G such that n is fixed by the decision procedure $fix_{DP}(n) = \mathsf{true}$. Assume that $\forall p \in Pred(n) : \neg fix_{DP}(p)$ and $\forall d \in Desc(n) : n \gg d$, then $fix_{\uparrow}(n) = \mathsf{true}$*

Proof. As n dominates all descendants, the decision procedure could have inferred that $n = FixVal_{DP}(n)$ by a chain of constant propagations either from the descendants in G of n or from its ancestors. Due to the definition of dominance, the constant propagation chain can enter the subgraph rooted at n only passing through n, or has to start in the subgraph and propagate upwards. According to our assumptions (Sec. 3), the decision procedure completely propagates constants. So, if the chain starts in some ancestor of n, at least one predecessor has to be fixed. If that's not the case, we can deduce that $n = FixVal_{DP}(n)$ must have been implied from the descendants of n.

Algorithm 2 Checking the Validity of VCs with Shared Structure. Function TRANSLATE translates the graph representation to a representation suitable for the decision procedure. SOLVE is the call to the decision procedure with the set of constraints C.

```
1:  clear table Fixed
2:  for each VC_i do
3:      C ← TRANSLATE(VC_i) ∪ VC_i = false
4:      for each n ∈ Desc(VC_i) do
5:          if n is a valid index into table Fixed then
6:              C ← C ∪ n = Fixed[n]
7:      status ← SOLVE(C)
8:      if status = satisfiable then
9:          Report bug
10:     FIX(VC_i, Fixed)
```

Theorem 1. *All of the expressions $n = FixVal_{DP}(n)$ computed by Alg. 1 are context-independent invariants.*

Proof. Follows directly from Lemma 1.

Fixed nodes n_i cannot be simply replaced with constants $FixVal_{DP}(n_i)$. Rather, one has to add constraint $n_i = FixVal_{DP}(n_i)$ as an additional constraint to the database of constraints that the decision procedure keeps. For instance, if $n = (a \leq 0)$ is a subgraph of one VC, and the algorithm fixes the value of n to true, and replaces node n with true, then some other VC which contains subgraph $m = n \wedge a > 0$ would be true as well, which is wrong. If only constraint $n =$ true is added to the constraint database, m becomes false. In other words, replacing the fixed nodes with constants is disallowed because it breaks dependencies among sub-expressions. Note that adding the additional constraint does not introduce inconsistency, because the same invariant would be inferred when the other VC is solved, but by adding it right away we speed-up the convergence and avoid redundant computation.

Finally, we give the overall algorithm (Alg. 2) to verify multiple VCs with sharing, as implemented in CALYSTO. Given a graph representation of a VC, the main loop first translates the graph into the form suitable for the given decision procedure, producing a set of constraints C, and negates the VC. For each node n whose value was fixed from below, the algorithm adds the corresponding constraint $n = FixVal_{DP}(n)$ to the set of constraints. The decision procedure is called with the set of constraints as a parameter. If the decision procedure finds the negated VC satisfiable, it reports a possible bug and continues. In the last step, Alg. 1 visits the nodes in the graph, and computes an approximation of the set of nodes whose values were fixed from below by the most recent call to the decision procedure, for use in solving subsequent VCs.

4.2 Example

In this section, we go through an example that is similar to what we have found in practice. The example illustrates expression sharing among VCs. Variables `a,b,c` are machine integers, and `s,t,u,v,y,x` are boolean variables. All operators used in the example are standard C-like operators.[6]

```
1  int f(int a, int b, bool s, bool t) {
2      if (a % 2) { a++; }
3      if (b % 2) { b++; }
4
5      int c = a * b;
6      int d = c & 3;
7      bool u = (d != 0);
8      bool v = (s == t);
9      bool y = (u || s);
10     bool x = (y || v);
11
12     if (x) {
13         assert(t); // VC1
14         ...
15     } else {
16         assert((a + b) % 2 == 0); // VC2
17         ...
18     }
19     ...
20 }
```

There are two assertions in the example: the first assertion can be violated, while the second cannot. Lines 2–3 increment odd numbers, so at line 5 both `a` and `b` are even. Thus, their product is a multiple of four. Therefore, the last two bits of the product will be zero, even in the case of an overflow. Hence, `d` is always zero.

In our implementation, the VCs are computed directly as maximally-shared graphs, as shown in Fig. 8, from the SSA [35] provided by the compiler front-end. A large part of the graph is shared. This sharing is especially valuable when expensive operations are shared, like multiplication. The computed graphs correspond to logical formulas in Fig. 7.

How would a SAT-based decision procedure handle these constraints? Each VC is solved independently of the others, and additional constraints are kept only for nodes fixed from below. We start solving `VC1` by adding the constraint $VC_1 =$ false. The decision procedure could deduce by constant propagation from the root: $x =$ true, $t =$ false, and those are all the invariants that can be found by trivial constant propagation. A typical SAT solver could continue with enumeration of possible solutions that would satisfy node `c`, which corresponds to the product of two conditionally defined variables. If `a` (resp. `b`) is odd, it will be incremented, so `a` (resp. `b`) is even at line 5. As mentioned previously, the least significant bit of even numbers is zero, so the two least significant bits of a product of even numbers are zero as well. Hence, the decision procedure eventually implies $d = 0$. By constant propagation it follows that $u =$ false. At that point, the decision procedure has to make another case split, and by setting $s =$ true, `VC1` is satisfied, meaning that the assertion can be violated. When `VC1` is being solved, node `u` dominates all leaves of its subgraph (each root node is solved independently, so VC2 still doesn't exist at this point). Node `u` was

[6] Operator % is the modulo operator, & is bitwise-and, || is logical-or, and ++ is post-increment.

$$VC1: \quad \Big(\Big(\Big(\big(\, ite(a_0\%2=0, a_0+1, a_0) * ite(b_0\%2=0, b_0+1, b_0)\,\big)\&3\Big)\neq 0\Big) \vee s \vee (s=t)\Big) \Rightarrow t$$

$$VC2: \quad \neg\Big(\Big(\Big(\big(\, ite(a_0\%2=0, a_0+1, a_0) * ite(b_0\%2=0, b_0+1, b_0)\,\big)\&3\Big)\neq 0\Big) \vee s \vee (s=t)\Big) \Rightarrow \Big(\Big(\big(\, ite(a_0\%2=0, a_0+1, a_0) + ite(b_0\%2=0, b_0+1, b_0)\,\big)\%2\Big)=0\Big)$$

Fig. 7. Two Verification Conditions. Corresponding maximally-shared graphs are shown in Fig. 8. The first VC is satisfiable, but not valid, while the second is valid.

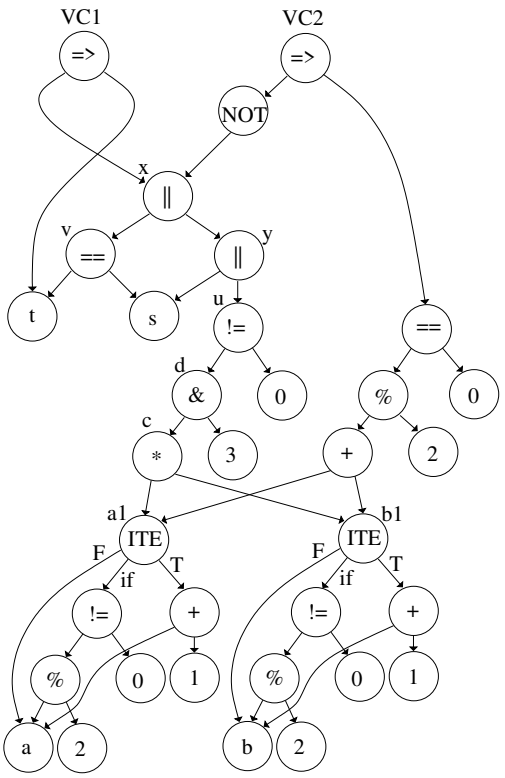

Fig. 8. Maximally-Shared Graph Representing Two Negated VCs. To simplify the graph layout, some constants are not shared. Edges of if-then-else (ITE) nodes are labelled with `if` for the condition branch, and `T` (resp. `F`) for true (resp. false) branches.

not fixed from above, but considering the subgraph rooted at `u`, the decision procedure was able to infer that $u = \mathsf{false}$. Since both conditions required by the Alg. 1 are met, `u` can be marked as fixed from below. Later, when `VC2` is constructed, the additional constraint $u = \mathsf{false}$ can be added to the set of constraints. Adding the constraint quickly prunes away most of the left branch of `VC2`, focusing the effort on the right branch. Since the sum of two even numbers is divisible by two, the right branch is true, meaning that $VC_2 = \mathsf{false}$ is unsatisfiable. Hence, the second assertion is valid.

5 Experimental Results

To test our approach, we used CALYSTO to generate VCs for six real-world, publicly-available C/C++ applications, ranging in size from 4 to 126 thousand lines of code (KLOC) before preprocessing. The benchmarks are the Bftpd ftp server, the Dspam spam filter, our boolean satisfiability solver HYPERSAT, the Licq ICQ chat client, the Wine Windows OS emulator, and the Xchat IRC client. For each program, for each pointer dereference, we generated a VC to check that the pointer is non-NULL (omitting VCs that were solved trivially by our expression simplifier). Although we demonstrate our approach on checking for NULL pointers, our method is independent of the property being verified, as long as the assumptions in Sec. 3 are met.

The experimental results are given in Table 1. The runtimes represent the time our SAT-based bit-vector arithmetic decision procedure SPEAR needed for solving all the VCs and include computation of the dominance relation. On only one of the smaller benchmarks, Licq, was the new approach somewhat slower. In all other cases, the new approach is faster. On Wine, the largest benchmark, the proposed approach speeds up the solving phase by 32%. There were also fewer timeouts with the new approach (meaning that the reported results are lower bounds on the speedup).

We attempted to understand why on some benchmarks (like Wine) our technique is very effective, while on others (like Licq) it is ineffective. Like any learning technique, our technique interacts with the solver's heuristics and therefore impacts the sequence of decisions made by the solver. The decision procedures community has tried to explain why certain decision heuristics work (and certain don't) for a long time (e.g., [36–38]), but still very little is known about heuristics, and frequently every researcher has his/her own interpretation. For those reasons, it is very difficult to give a definitive qualitative and quantitative analytical answer about when to apply our technique. The decision should be made empirically. Our conjecture is, though, that applications that are implemented in a modular fashion, with well-defined narrow interfaces that are often used, are a good candidate for using our technique, as it is likely that our technique will be able to learn context-independent facts about those interfaces.

The key question is whether the derived context-independent invariants are able to accelerate the solver enough to overcome the cost of deriving them. We profiled

Benchmark	KLOC	#VCs	Base Approach		New Approach	
			Time (sec)	Timeouts	Time (sec)	Timeouts
Bftpd v1.6	4	1130	725.8	0	582.5	0
HyperSAT v1.7	9	1363	5.3	0	5.1	0
Licq v1.3.4	20	2009	199.6	0	214.5	0
Dspam v3.6.5	37	8627	3478.6	8	3157.6	6
Xchat v2.6.8	76	8090	368.5	0	365.8	0
Wine v0.9.27	126	9000	1881.4	2	1266.7	0

Table 1. The first column gives the name and version of the benchmark. KLOC is the number of source code lines, in thousands, before preprocessing. #VCs is the number of checked VCs. As is typical, almost all VCs are UNSAT, since satisfiable VCs correspond to bug reports. The next four columns give the total VC checking time in seconds (including timeouts) and the number of timeouts, for the base approach (i.e., the same system without the newly proposed method) vs. the newly proposed method. The timeout limit was 300 secs. Experiments were on a dual-processor AMD X2 4600+ machine with 2 GB RAM, running Linux 2.6.15. Memory consumption was not a bottleneck on any of the benchmarks.

the runtimes of CALYSTO on the benchmarks in Table 1 and found that the computation of nodes fixed from below was dwarfed by the solving time and did not even show in the profile data.

Our decision procedure, SPEAR, was already highly optimized, and features several techniques (like abstraction, lazy interpretation [23], gate-optimal VC encoding, and several others) that result in significant performance improvements over a standard, direct "bit-blasting" translation of the VCs into SAT. The results presented in Table 1 clearly show that exploiting shared structure can push a state-of-the-art static checker even further.

6 Related Work

We know of no closely related work on exploiting shared structure between multiple verification conditions. However, there is a large body of work that influenced, or could be influenced by, this work.

Static Checking. Our work on CALYSTO is in the tradition of extended static checking [11]. In particular, CALYSTO was inspired by ESC/Java [39], Saturn [13], and Boogie [40]. Our overall goal was to provide complete automation, yet maintain the precision of a bit-accurate software model checker like CBMC [41], while matching or exceeding the scalability of static checkers like Boogie or Saturn.

Previous works (e.g., [40,39]) often construct only one VC per function, and analyze each function independently of others (this is possible only if users provide pre- and post-conditions for every function). In this usage mode, it is unlikely that anything except pre- and post- conditions will be shared. While the total number of VCs is smaller, this approach reports only one possible assertion violation per function. In contrast, software compilers tend to do extensive error recovery (e.g., [42]) to report as many errors as possible, so that programers can fix multiple bugs in each, potentially time consuming, compilation cycle. Our usage model accepts the same philosophy by analyzing a larger number of VCs and reporting multiple error violations at once.

Verification Conditions. Traditionally, VCs are computed by Dijkstra's weakest precondition transformer [43], as is done for example in ESC/Java and Boogie. A naïve representation of VCs computed by the weakest precondition can be exponential in the size of the code fragment being checked, but this blow-up can be avoided by the introduction of fresh variables to represent intermediate expressions [27,1,2]. Equivalently, we keep the formulas in the form of maximally shared graphs, making structural reasoning easier, as illustrated in this paper. This representational difference is otherwise insignificant.

What sets this paper apart from previous work on VCs is our focus on exploiting common sub-expressions shared among multiple VCs. We explore how much we can learn from solving a set of VCs and how we can apply that knowledge to solve the remaining VCs more efficiently.

Learning. Our contribution can be viewed as an automatic learning technique. Given a set of VCs, the technique learns from the implicants that a decision procedure implied, and attempts to reuse that knowledge later if the remaining VCs share some sub-expressions with the already solved ones.

Learning is an efficient technique for speeding up decision procedures, and has been especially effective in boolean satisfiability (SAT) solvers [44]. The new aspect of the problem that we are considering is *context-dependence* — facts learned about a shared subgraph while solving one VC might not hold in the context of others.

Stump and Dill [45] proposed context-dependent caching and proof compression for an Edinburgh LF decision procedure, but they considered caching only for subgraphs of a single formula and did not consider sharing between multiple formulas. While solving each individual VC, our static checker CALYSTO already eliminates common sub-expressions, and our bit-vector arithmetic decision procedure SPEAR features its own *intra-VC* learning (caching) mechanism. In contrast, the contribution of the present paper is *inter-VC* learning.

While solving even a single VC, decision procedures can produce a large number of context-dependent implicants. In practice, a small percentage of them end up being useful, so aggressive implicant deletion strategies [46] are needed.

These experimental results motivated us to focus on simple context-independent implicants of the form of equality with a constant.

Incremental Decision Procedures. Our contribution can also be viewed as a type of incremental decision procedure, where the goal is to solve multiple, related problem instances while reusing information from one instance to another. The early work of Hooker [47] considered incremental Boolean satisfiability in a simplistic setting, without learning and clause deletion. Strichman [17], in his generalization of the work by Silva and Sakallah [16], noticed that in the context of bounded model checking, the clauses that depend only on the structure of the model remain valid when the unrolling depth is increased. This was one of the first applications of incremental satisfiability. Since Eén and Sörensson implemented efficient incremental interface in their MiniSAT solver [18], incremental satisfiability has found a large number of applications in various domains. Some automated theorem provers, like Yices [5] and CVC [48], feature push/pop commands that allow undoing logical reasoning since the last checkpoint (push). These push/pop techniques are not applicable to our problem: In order to avoid the worst-case exponential cost of interprocedural path-sensitivity and context-sensitivity, CALYSTO performs structural abstraction [23] of VCs, which means it is not known *a priori* which nodes will end up being shared, so every single sub-expression would need to be pushed as a new context. So, the push/pop commands are not a viable solution to our problem.

Structure Exploitation. Many researchers have looked into how to exploit structure for more efficient verification. Rountev at al. [49] observed that large libraries change less frequently than the applications that use them, so the libraries can be pre-analyzed for speeding up verification of the applications. Conway et al. [50] observed that programs are usually modified in small incremental steps. So, after the application was verified once, only the modified functions and functions that transitively call them have to be re-verified. Within a single codebase, our previous work [23] showed how the structure of a single interprocedural verification condition can be exploited at a function-level granularity. In this paper, we explore a new dimension of the problem that has not (to the best of our knowledge) been explored before. Namely, we are interested in elimination of redundancy at a finer level of granularity — individual expressions. This redundancy is inherent to any software verification technique simply because a large majority of execution paths share some common sequence of statements. Our technique is orthogonal to the above mentioned approaches, and can be combined with them.

7 Future Work

It would be useful to improve the quality of approximation of the set of nodes fixed from below, while maintaining the low computational cost. Since we observed more structure-sharing in practice than our technique is able to exploit, we believe that improvements in that direction could provide even more significant speedups.

Finding more expressive context-independent invariants could also boost the performance of static checking. Such context-independent learning would probably run into similar problems as learning in decision procedures — which implicants to keep and for how long. Considering that learning has proven itself in SAT solvers as an indispensable technique without which no solver today is competitive, we believe that this direction is particularly promising.

We have focused on the case where VCs are solved one-by-one. If multiple VCs are available all at once, solving the VCs in a different, heuristically-chosen order might allow deriving more context-independent invariants. Furthermore, it should be possible to analyze the maximally shared graph to quickly find the shared subgraphs between the multiple VCs. Only these nodes need to be considered as candidates to be context-independent invariants, reducing the overhead of our approach.

8 Conclusion

We have demonstrated a novel way to exploit shared, expression-level structure available in verification conditions. The approach relies on simple invariants inferred by automatic decision procedures. The proposed technique computes a subset of those invariants which can be used safely in a context-independent manner. Our experimental results demonstrate that the technique can substantially improve the performance of static checking. As scalability is the primary limitation of automatic software verification tools, these results are a step towards more widely applicable, practical formal verification of software.

References

1. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Proceedings of the 28^{th} ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'01). Volume 36., New York, NY, USA, ACM Press (2001) 193–205
2. Leino, K.R.M.: Efficient weakest preconditions. Information Processing Letters **93** (2005) 281–288
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming language design and implementation (PLDI'01). Volume 36., New York, NY, USA, ACM Press (2001) 203–213
4. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding bit-vector arithmetic with abstraction. In Grumberg, O., Huth, M., eds.: Proceedings of the 13^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07). Volume 4424 of Lecture Notes in Computer Science (LNCS)., Springer (2007) 358–372

5. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In Ball, T., Jones, R.B., eds.: Proceedings of the 18^{th} International Conference on Computer Aided Verification (CAV'06). Volume 4144 of Lecture Notes in Computer Science (LNCS)., Springer (2006) 81–94
6. Ball, T., Lahiri, S.K., Musuvathi, M.: Zap: Automated theorem proving for software analysis. In Sutcliffe, G., Voronkov, A., eds.: Proceedings of the 12^{th} International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05). Volume 3835 of Lecture Notes in Computer Science (LNCS)., Springer (2005) 2–22
7. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In Alur, R., Peled, D.A., eds.: Proceedings of the 16^{th} International Conference on Computer Aided Verification (CAV'04). Volume 3114 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag (2004) 175–188
8. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38^{th} Design Automation Conference (DAC'01), ACM Press (2001) 530–535
9. Nelson, G.: Techniques for program verification. PhD thesis, Stanford University, Stanford, California, USA (1980)
10. Johnson, S.: Lint, a C Program Checker. Technical Report 65, Bell Laboratories (1977)
11. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Technical Report SRC-RR-159, Compaq Systems Research Center, Palo Alto, California, USA (1998) Now available from HP Labs.
12. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of the 4^{th} conference on Symposium on Operating System Design & Implementation (OSDI'00), Berkeley, California, USA, USENIX Association (2000) 1–1
13. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Proceedings of the 32^{nd} ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'05). Volume 40., New York, NY, USA, ACM Press (2005) 351–363
14. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In Kozen, D., ed.: Workshop on Logics of Programs. (1981) 52–71 Published 1982 as Lecture Notes in Computer Science Number 131.
15. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: 5th International Symposium on Programming, Springer (1981) 337–351 Lecture Notes in Computer Science Number 137.
16. Silva, J.P.M., Sakallah, K.A.: Robust search algorithms for test pattern generation. In: Proceedings of the 27^{th} International Symposium on Fault-Tolerant Computing (FTCS '97), Washington, DC, USA, IEEE Computer Society (1997) 152
17. Shtrichman, O.: Pruning Techniques for the SAT-Based Bounded Model Checking Problem. In: Proceedings of the 11^{th} IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01), London, UK, Springer-Verlag (2001) 58–70
18. Eén, N., Sörensson, N.: An Extensible SAT-solver. (2004) 502–518
19. Babić, D., Hu, A.J.: Calysto: Scalable and precise extended static checking. In: 30th International Conference on Software Engineering (ICSE'08). (2008) 211–220
20. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. Journal of ACM **50** (2003) 63–69
21. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13** (1991) 451–490
22. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
23. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. In Damm, W., Hermanns, H., eds.: Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV'07). Volume 4590 of Lecture Notes in Computer Science (LNCS)., Springer (2007) 371–383
24. Babić, D.: Exploiting Structure for Scalable Software Verification. PhD thesis, University of British Columbia, Computer Science Department (2008)
25. Ball, T., Rajamani, S.K.: SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research (2001)
26. Babić, D., Musuvathi, M.: Modular Arithmetic Decision Procedure. Technical Report MSR-TR-2005-114, Microsoft Research Redmond (2005)
27. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In Siekmann, J., Wrightson, G., eds.: Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970. Springer, Berlin, Heidelberg (1983) 466–483
28. Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. In Bacchus, F., Walsh, T., eds.: Proceedings of the 8^{th} International Conference on Theory and Applications of Satisfiability Testing (SAT'05). Volume 3569 of Lecture Notes in Computer Science (LNCS)., Springer (2005) 61–75
29. Eén, N., Sörensson, N.: An extensible SAT solver. In: Proceedings of the 6^{th} International Conference on theory and Applications of Satisfiability Testing (SAT'03). Volume 2919 of Lecture Notes in Computer Science (LNCS)., Santa Margherita Ligure, Italy, Springer (2003) 502–518
30. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5** (1962) 394–397
31. Hutter, F., Babić, D., Hoos, H.H., Hu, A.J.: Boosting Verification by Automatic Tuning of Decision Procedures. In: Proceedings of the Formal Methods in Computer Aided Design (FMCAD'07), Washington, DC, USA, IEEE Computer Society (2007) 27–34
32. Prosser, R.T.: Applications of boolean matrices to the analysis of flow diagrams. In: Proceedings of the 16^{th} Eastern Joint Computer Conference, New York, Spartan Books (1959) 133–138
33. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems (TOPLAS) **1** (1979) 121–141
34. Detlefs, D., Nelson, G., Saxe, J.S.: Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto (2003)
35. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15^{th} ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'88), New York, NY, USA, ACM Press (1988) 12–27
36. Bhalla, A., Lynce, I., de Sousa, J., Marques-Silva, J.: Heuristic backtracking algorithms for SAT. In: Proceedings of the 4^{th} In-

ternational Workshop on Microprocessor Test and Verification (MTV'03), Austin, Texas, USA (2003) 69–74
37. Silva, J.P.M.: The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In: Proceedings of the 9^{th} Portuguese Conference on Artificial Intelligence (EPIA'99). Volume 1695 of Lecture Notes in Computer Science (LNCS)., Springer (1999) 62–74
38. Shacham, O., Zarpas, E.: Tuning the VSIDS Decision Heuristic for Bounded Model Checking. In: Proceedings of th 4^{th} International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV'03), IEEE Computer Society (2003) 75–79
39. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02). Volume 37., New York, NY, USA, ACM Press (2002) 234–245
40. Leino, K.R.M., Müller, P.: A verification methodology for model fields. In Sestoft, P., ed.: Proceedings of the 15^{th} European Symposium on Programming (ESOP'06), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS'06). Volume 3924 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag (2006) 115–130
41. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of the 40^{th} conference on Design automation (DAC'03), New York, NY, USA, ACM Press (2003) 368–371
42. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA (1986)
43. Dijkstra, E.W., Scholten, C.S.: Predicate calculus and program semantics. Springer-Verlag New York, Inc., New York, NY, USA (1990)
44. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD'01), Piscataway, New Jersey, USA, IEEE Press (2001) 279–285
45. Stump, A., Dill, D.L.: Faster proof checking in the edinburgh logical framework. In Voronkov, A., ed.: Proceedings of the 18^{th} International Conference on Automated Deduction (CADE'02). Volume 2392 of Lecture Notes in Computer Science (LNCS)., London, UK, Springer-Verlag (2002) 392–407
46. Mahajan, Y.S., Fu, Z., Malik, S.: Zchaff2004: An efficient SAT solver. In: Proceedings of the 7^{th} International Conference on Theory and Applications of Satisfiability Testing (SAT'04). Volume 3542 of Lecture Notes in Computer Science (LNCS). (2004) 360–375
47. Hooker, J.N.: Solving the incremental satisfiability problem. Journal of Logic Program. **15** (1993) 177–186
48. Stump, A., Barrett, C.W., Dill, D.L.: CVC: A Cooperating Validity Checker. In: Proceedings of the 14^{th} International Conference on Computer Aided Verification (CAV'02). Volume 2404 of Lecture Notes in Computer Science (LNCS)., London, UK, Springer-Verlag (2002) 500–504
49. Rountev, A., Kagan, S., Marlowe, T.J.: Interprocedural dataflow analysis in the presence of large libraries. In Mycroft, A., Zeller, A., eds.: Proceedings of the 15^{th} International Conference on Compiler Construction (CC'06), held as a part of the Joint European Conferences on Theory and Practice of Software (ETAPS'06). Volume 3923 of Lecture Notes in Computer Science (LNCS)., Springer (2006) 2–16
50. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In Etessami, K., Rajamani, S.K., eds.: Proceedings of the 17^{th} International Conference on Computer Aided Verification (CAV'05). Volume 3576 of Lecture Notes in Computer Science (LNCS)., Berlin, Germany, Springer (2005) 449–461