Exploiting Shared Structure in Software Verification Conditions

Domagoj Babić and Alan J. Hu

Computer Science Department
University of British Columbia

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, propose a novel and efficient approach to exploit this sharing, and provide experimental results that this approach can significantly improve the performance of verification, even on path- and context-sensitive and dataflow-intensive properties.

1 Introduction

Verification conditions (VCs) are logical formulas, constructed from a system and desired correctness properties, such that the validity of verification conditions corresponds to the correctness of the system. 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. [11, 15]), abstraction to make the VCs simpler for the decision procedure (e.g. [4, 5]), and the efficiency of the decision procedures themselves (e.g. [9, 3, 12, 19, 20]).

In our previous work [1], we showed how the structure of a single interprocedural verification condition can be exploited at a coarse function level. 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, propose an algorithm for exploiting this shared structure, and provide experimental evidence that our approach can cut runtime by almost one third and reduce the number of timeouts.

1.1 Background and Related Work

Static Checking The work in this paper fits in the context of static checking of software. The distinction between static checking and model checking is 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. Our overall goal is to maintain the precision of a bit-accurate software model checker like CBMC [14], while matching or exceeding the scalability of static checkers like Boogie [17] or Saturn [27].

We use our static checker CALYSTO, but the contribution of this paper can be applied to any static checker that uses a decision procedure, assuming some reasonable properties of VCs (see Sec. 2). Boogie and Saturn are the closest relatives of CALYSTO. Boogie is a mature tool that performs *intraprocedural* analysis and requires user-provided function/class interface invariants. Boogie uses abstract interpretation to compute sound invariants of certain types of loops found in programs, while others are unrolled and terminated with an assumption that the loop test is false [16]. CALYSTO is less mature and handles loops either by unrolling them (unsound) as in ESC/Java [10] or by considering all loop-carried values unconstrained (sound). Standard, more precise loop invariant computation techniques can be used to replace loops with loop invariants, as a CALYSTO-preprocessing technique. The most significant difference is that CALYSTO requires no user-provided interface invariants. Instead, CALYSTO performs path- and context-sensitive *interprocedural* analysis. Such analysis is inherently more expensive than the intraprocedural analysis in Boogie, so we focus on exploiting structure at various levels of granularity to achieve scalability. For instance, in our previous work [1], we showed how structure can be exploited to avoid the exponential blowup of context-sensitive analysis in many cases. Saturn is path-sensitive, but performs only partially context-sensitive analysis by computing summaries as projections onto a set of predicates. CALYSTO, on the other hand, is fully context sensitive, which means that it can handle dataflow-related properties more precisely. Saturn demonstrated that SAT solvers can be used to prove VCs, but it uses off-the-shelf SAT solvers. In our experience, we have found that tight integration of the static checker with a custom-tailored decision procedure offers significant performance improvements, hence our research on exploiting structural properties of VCs by the decision procedures.

Verification Conditions Traditionally, VCs are computed by Dijkstra's weakest precondition transformer [8], as is done for example in ESC/Java [10] 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 [26, 11, 15]. Equivalently, we can keep the formulas in the form of graphs that correspond to the abstract syntax trees of the parsed formulas, with common sub-expressions shared. Such graphs make structural reasoning easier, so we shall use the graph representation in this paper. This representational difference is otherwise insignificant.

Two things set our research apart from previous work on VCs. First, as mentioned above, we do not assume user-provided interface invariants, but rather perform context-

sensitive interprocedural analysis. Second, we focus on exploiting common subexpressions shared among multiple VCs. Our goal is to 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 subexpressions 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 [28]. 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 [25] 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 subexpressions, and our SAT-based decision procedure SPEAR features its own *intra-VC* learning (caching) mechanism. In contrast, the contribution of the present paper is *inter-VC* learning.

Structure Exploitation Many researchers have looked into how to exploit structure for more efficient verification. Starting from the coarsest level of granularity, Rountev at al. [23] 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. [6] 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. Our work explores 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.

2 Preliminaries

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.

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 modular (machine bit-vector) arithmetic. Our decision procedure SPEAR[2] is

[1] For instance, the 2004 JPEG security exploit (see e.g. [2]).

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

based on a SAT solver and supports all standard modular arithmetic operators on finite bit-vectors, including expensive operators (like multiplication and division). Although we use modular arithmetic, the contribution is largely independent of the chosen logic.

When automated decision procedures are used for proving VCs, the validity of a verification condition VC is usually being 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.

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

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 $Desc(n)$.

To analyze the shared subgraphs, we rely upon the dominance relation [21]:

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 $n \underline{\geqslant} m$. If $n \neq m$, n strictly dominates m, denoted $n \gg m$.

The dominance relation is a partial order (reflexive, antisymmetric, and transitive) and can be computed in $\mathscr{O}(N\alpha(E,N))$ [18] time, where α is the extremely slowly growing inverse of Ackermann's function. In practice, a simpler $\mathscr{O}(E \log N)$ algorithm [18] 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, as mentioned already, we assume that the VCs are representable by acyclic graphs corresponding to abstract syntax trees obtained by parsing the formula. Most software static checking tools (including Saturn, ESC/Java, Boogie, and CALYSTO) produce VCs that have such structure. An example of a graph representation of two VCs that share some subgraphs is shown in Fig. 1.

Second, 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 [20] framework generate conjunctions of equalities (providing that the individual theories are convex), and it is easy to extract the equalities that satisfy our requirement.

Third, 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 [7], on which the Nelson-Oppen framework is based, satisfy this requirement.

Fourth, we assume that the proper subexpressions 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. 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. 1. 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. 1 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 constraint *root_node* = false and then check 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.

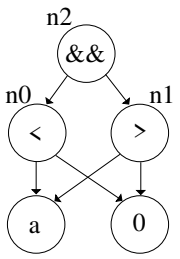

Fig. 1. Small maximally-shared graph representing $a < 0 \wedge a > 0$. Successors of non-commutative operators are ordered in the natural order (from left to right). Operator nodes are labelled with the operator (inscribed) and the name of corresponding variable used in renaming (adjacent to the node).

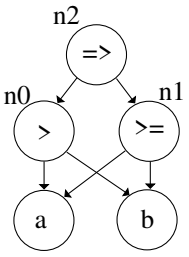

Fig. 2. Graph corresponding to the set of constraints in Example 2.

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 $n_2 = \mathsf{false}$ to the set, forcing the output to false. The set of constraints becomes unsatisfiable, meaning that the original VC was valid. □

If the consistency assumption were violated, then the decision procedure could imply arbitrary implicants, because false can imply anything. The consistency assumption ensures that the implicants derived from a subexpression are meaningful.

3 Exploiting Shared Structure

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

A direct approach is to construct a 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 that disjunct from further consideration. Everything that the theorem prover learns can be re-used, 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. Instead, we seek to distill out implicants learned while solving one VC that are useful for solving another VC. However, 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. 2 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 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 subexpression 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 subexpression 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. 2) 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. *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. *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 [20][3]. 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.

It is worth noting that simple incrementality [13] cannot be used for handling multiple contexts. When the contexts are changed, assumptions and their implicants unrelated to the new context have to be removed, so the implication graphs have to be recorded — exactly what we are trying to avoid. Some automated theorem provers, like Yices [9] and CVC [24], feature push/pop commands that allow undoing logical reasoning since the last checkpoint (push). Even with these commands, we would need to push a new context for each potentially shared node, which would be prohibitively expensive. Furthermore, if lazy construction of VCs is used, then it is not known *a priori* which nodes will end up being shared, so every single subexpression would need to be pushed as a new context.

3.1 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 lazy structural abstraction [1], 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 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

[3] 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.

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)
```

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.

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: (1) The invariant was implied from below, relying only on the descendants of n. (2) The invariant was implied from above, possibly all the way from the root node. (3) The constant propagation chain came from above, avoiding n, fixed the value of some descendant of n, which in turn implied the invariant. 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. 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 first contains n_1, 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 $n_i = \mathit{FixVal}_{DP}(n_i)$ 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. 4, 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 $\mathit{fix}_{DP}(n) = \mathsf{true}$. Assume that $\forall p \in \mathit{Pred}(n) : \neg\mathit{fix}_{DP}(p)$ and $\forall d \in \mathit{Desc}(n) : n \gg d$, then $\mathit{fix}_{\uparrow}(n) = \mathsf{true}$*

Proof. As n dominates all descendants, the decision procedure could have inferred that $n = \mathit{FixVal}_{DP}(n)$ 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. 2), the decision procedure completely propagates constants. So, if the 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 $n = \mathit{FixVal}_{DP}(n)$ must have been implied from the descendants of n.

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

Proof. Follows from Lemma 1

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 $n = \mathit{FixVal}_{DP}(n)$ 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 subsquent VCs.

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)
```

3.2 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.[4]

```
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 can't. 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. 3, from the SSA [22] provided by the compiler front-end. A large part

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

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.

of the graph is shared. This sharing is especially valuable when expensive operations are shared, like multiplication.

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 $VC_1 = \mathsf{false}$. The decision procedure could deduce by constant propagation from the root: $x = \mathsf{true}, t = \mathsf{false}$, 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 $u = \mathsf{false}$. At that point, the decision procedure has to make another case split, and by setting $s = \mathsf{true}$, `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 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.

4 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

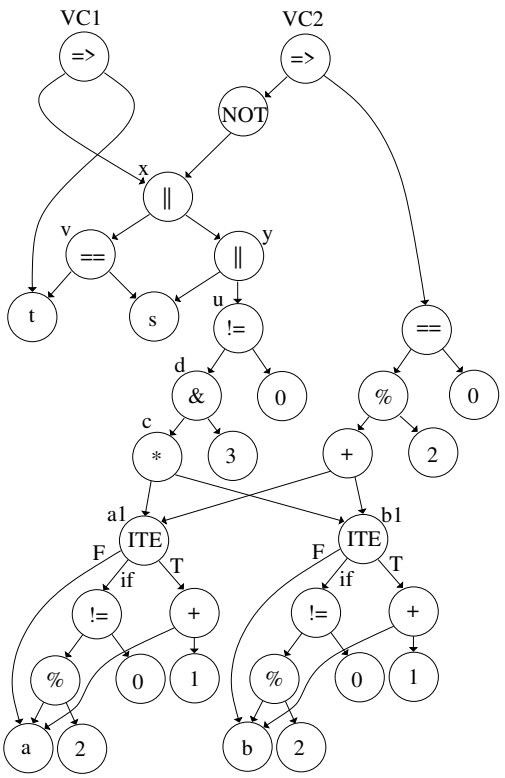

Fig. 3. Maximally-shared graph representing two negated VCs. To simplify the graph layout, some constants are not shared. Edges of if-then-else (ITE) nodes are labelled with `if` for the condition branch, and `T` (resp. `F`) for true (resp. false) branches.

(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. 2 are met.

The experimental results are given in Table 1. The runtimes represent the time our SAT-based modular 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).

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

that the overhead of our approach is very low, yet in some cases, it provides a substantial speedup. SPEAR was already highly optimized, and features several techniques (like abstraction, lazy interpretation [1], 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 show that exploiting shared structure can push a state-of-the-art static checker even further.

5 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.

6 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. D. Babić and A. J. Hu. Structural abstraction of software verification conditions. *CAV*, volume 4590 of *LNCS*, pages 371–383, Berlin, 2007. Springer. To appear.
2. D. Babić and M. Musuvathi. Modular Arithmetic Decision Procedure. Technical Report TR-2005-114, Microsoft Research Redmond, 2005.
3. T. Ball, S. K. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. In *LPAR*, volume 3835 of *LNCS*, pages 2–22. Springer, 2005.

4. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213, 2001.
5. R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *TACAS*, volume 4424 of *LNCS*, pages 358–372. Springer, 2007.
6. C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In *CAV*, volume 3576 of *LNCS*, pages 449–461, Berlin, 2005. Springer.
7. D. Detlefs, G. Nelson, and J. S. Saxe. Simplify: A Theorem Prover for Program Checking. Technical report, HP Laboratories Palo Alto, Technical Report HPL-2003-148, 2003.
8. E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
9. B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, ACM SIGPLAN Notices, pages 234–245, 2002.
11. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205. ACM Press, 2001.
12. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *CAV*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
13. J. N. Hooker. Solving the incremental satisfiability problem. *J. Log. Program.*, 15(1-2):177–186, 1993.
14. D. Kroening, E. Clarke, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *DAC*, pages 368–371. ACM Press, 2003.
15. K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
16. K. R. M. Leino and F. Logozzo. Loop invariants on demand. In K. Yi, editor, *APLAS*, volume 3780 of *LNCS*, pages 119–134. Springer, 2005.
17. K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *ESOP*, volume 3924 of *LNCS*, pages 115–130. Springer-Verlag, 2006.
18. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.
19. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM Press, 2001.
20. G. Nelson. *Techniques for program verification*. PhD thesis, Stanford University, 1979.
21. R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138. Spartan Books, 1959.
22. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL*, pages 12–27. ACM Press, 1988.
23. A. Rountev, S. Kagan, and T. J. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In A. Mycroft and A. Zeller, editors, *CC*, volume 3923 of *LNCS*, pages 2–16. Springer, 2006.
24. A. Stump, C. Barrett, and D. Dill. CVC: A Cooperating Validity Checker. In *CAV*, 2002.
25. A. Stump and D. L. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In A. Voronkov, editor, *CADE*, volume 2392 of *LNCS*, pages 392–407. Springer, 2002.
26. G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, 1983.
27. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363. ACM Press, 2005.
28. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, pages 279–285. IEEE Press, 2001.