Small Formulas for Large Programs: On-line Constraint Simplification in Scalable Static Analysis

Isil Dillig Thomas Dillig Alex Aiken
{isil, tdillig, aiken}@cs.stanford.edu

Department of Computer Science, Stanford University

Abstract. Static analysis techniques that represent program states as formulas typically generate a large number of redundant formulas that are incrementally constructed from previous formulas. In addition to querying satisfiability and validity, analyses perform other operations on formulas, such as quantifier elimination, substitution, and instantiation, most of which are highly sensitive to formula size. Thus, the scalability of many static analysis techniques requires controlling the size of the generated formulas throughout the analysis. In this paper, we present a practical algorithm for reducing SMT formulas to a *simplified form* containing no redundant subparts. We present experimental evidence that on-line simplification of formulas dramatically improves scalability.

1 Introduction

Software verification techniques have benefited greatly from recent advances in SAT and SMT solving by encoding program states as formulas and determining the feasibility of these states by querying satisfiability. Despite tremendous progress in solving SAT and SMT formulas over the last decade [1–5], the scalability of many software verification techniques relies crucially on controlling the size of the formulas generated by the analysis, because many of the operations performed on these formulas are highly sensitive to formula size. For this reason, much research effort has focused on identifying only those states and predicates relevant to some property of interest. For example, *predicate abstraction*-based approaches using *counter-example guided abstraction refinement* [6–8] attempt to discover a small set of predicates relevant to verifying a property and only include this small set of predicates in their formulas. Similarly, many path-sensitive static analysis techniques have successfully employed various heuristics to identify which path conditions are likely to be relevant for some property of interest. For example, *property simulation* only tracks those branch conditions for which the property-related behavior differs along the arms of the branch [9]. Other path-sensitive analysis techniques attempt to improve their scalability by either only tracking path conditions intraprocedurally or by heuristically selecting a small set of predicates to track across function boundaries [10, 11].

All of these different techniques share one important underlying assumption that has been validated by a large body of empirical evidence: Many program conditions do not matter for verifying most properties of interest, making it possible to construct much smaller formulas sufficient to prove the property. If this is indeed the case, then one might suspect that even if we construct a

formula ϕ characterizing some program property P without being particularly careful about what conditions to track, it should be possible to use ϕ to construct a much smaller, equivalent formula ϕ' for P since many predicates used in ϕ do not affect its truth value.

In this paper, we present a systematic and practical approach for simplifying formulas that identifies and removes irrelevant predicates and redundant subexpressions as they are generated by the analysis. In particular, given an input formula ϕ, our technique produces an equivalent formula ϕ' such that no simpler equivalent formula can be obtained by replacing any subset of the *leaves* (i.e., syntactic occurrences of atomic formulas) used in ϕ' by *true* or *false*. We call such a formula ϕ' *simplified*.

Like all the afore-mentioned approaches to program verification, our interest in simplification is motivated by the goal of generating formulas small enough to make software verification scalable. However, we attack the problem from a different angle: Instead of restricting the set of predicates that are allowed to appear in formulas, we continuously simplify the constraints generated by the analysis. This approach has two advantages: First, it does not require heuristics to decide which predicates are relevant, and second, this approach removes all redundant subparts of a formula in addition to filtering out irrelevant predicates.

To be concrete, consider the following code snippet:

```
enum op_type {ADD=0, SUBTRACT=1, MULTIPLY=2, DIV=3};
int perform_op(op_type op, int x, int y) {
   int res;
   if(op == ADD) res = x+y;
   else if(op == SUBTRACT) res = x-y;
   else if(op == MULTIPLY) res = x*y;
   else if(op == DIV) { assert(y!=0); res = x/y; }
   else res = UNDEFINED;
   return res; }
```

The `perform_op` function is a simple evaluation procedure inside a calculator program that performs a specified operation on `x` and `y`. This function aborts if the specified operation is division and the divisor is 0. Assume we want to know the constraint under which the function returns, i.e., does not abort. This constraint is given by the disjunction of the constraints under which each branch of the `if` statement does not abort. The following formula, constructed in a straightforward way from the program, describes this condition:

$$op = 0 \vee (op \neq 0 \wedge op = 1) \vee (op \neq 0 \wedge op \neq 1 \wedge op = 2) \vee$$
$$(op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op = 3 \wedge y \neq 0) \vee$$
$$(op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op \neq 3)$$

Here, each disjunct is associated with one branch of the `if` statement. In each disjunct, a disequality constraint of the form $op \neq 0, op \neq 1, \ldots$ states that the previous branches were not taken, encoding the semantics of an `else` statement. In the fourth disjunct, the additional constraint $y \neq 0$ encodes that if this branch is taken, `y` cannot be 0 for the function to return.

While this automatically generated constraint faithfully encodes the condition under which the function returns, it is far from concise. In fact, the above

constraint is equivalent to the much simpler formula:

$$op \neq 3 \vee y \neq 0$$

This formula is in *simplified form* because it is equivalent to the original formula and replacing any of the remaining leaves by *true* or *false* would not result in an equivalent formula. This simpler constraint expresses exactly what is relevant to the function's return condition and makes no reference to irrelevant predicates, such as $op = 0, op = 1$, and $op = 2$. Although the original formula corresponds to a brute-force enumeration of all paths in this function, its simplified form yields the most concise representation of the function's return condition without requiring specialized techniques for identifying relevant predicates.

The rest of the paper is organized as follows: Section 2 introduces preliminary definitions. Section 3 defines simplified form and highlights some of its properties. Section 4 presents a practical simplification algorithm, and Section 5 describes simplification in the context of program analysis. Section 6 reports experimental results, and Section 7 surveys related work. To summarize, this paper makes the following key contributions:

- We propose *on-line constraint simplification* as an effective technique for controlling the size of formulas generated while analyzing a program.
- We define what it means for a formula to be in *simplified form* and detail some important properties of this form.
- We give a practical algorithm for reducing formulas to their simplified form and show how this algorithm naturally integrates into the DPLL($\mathcal{T}$) framework for solving SMT formulas.
- We demonstrate the effectiveness of our on-line simplification algorithm in the context of a program verification framework and show that simplification improves overall performance by orders of magnitude, often allowing analysis runs that did not terminate within the allowed resource limits to complete in just a few seconds.

2 Preliminaries

Any quantifier-free formula $\phi_{\mathcal{T}}$ in theory $\mathcal{T}$ is defined by the following grammar:

$$\phi_{\mathcal{T}} := \mathit{true} \mid \mathit{false} \mid A_{\mathcal{T}} \mid \neg A_{\mathcal{T}} \mid \phi'_{\mathcal{T}} \wedge \phi''_{\mathcal{T}} \mid \phi'_{\mathcal{T}} \vee \phi''_{\mathcal{T}}$$

In the above grammar, $A_{\mathcal{T}}$ represents an *atomic formula* in theory $\mathcal{T}$, such as the boolean variable x in propositional logic or the inequality $a + 2b \leq 3$ in linear arithmetic. Observe that the above grammar requires formulas to be in *negation normal form (NNF)* because only atomic formulas may be negated. While the rest of this paper relies on formulas being in NNF, this restriction is not important since any formula may be converted to NNF using DeMorgan's laws in linear time without increasing the size of the formula (see Definition 2).

Definition 1. (Leaf) *We refer to each occurrence of an atomic formula $A_{\mathcal{T}}$ or its negation $\neg A_{\mathcal{T}}$ as a* leaf *of the formula in which it appears.*

It is important to note that different occurrences of the same (potentially negated) atomic formula in $\phi_{\mathcal{T}}$ form distinct leaves. For example, the two occurrences of $f(x) = 1$ in $f(x) = 1 \vee (f(x) = 1 \wedge x + y \leq 1)$ correspond to two distinct leaves.

In the rest of this paper, we restrict our focus to quantifier-free formulas in theory $\mathcal{T}$, and we assume there is a decision procedure $D_{\mathcal{T}}$ that can be used to decide the satisfiability of a quantifier-free formula $\phi_{\mathcal{T}}$ in theory $\mathcal{T}$. Where irrelevant, we omit the subscript $\mathcal{T}$ and denote formulas by ϕ.

Definition 2. (Size) *The size of a formula ϕ is the number of leaves ϕ contains.*

Definition 3. (Fold) *The fold operation removes constant leaves (i.e.,* true*,* false*) from the formula. In particular,* Fold(ϕ) *is a formula ϕ' such that (i) $\phi \Leftrightarrow \phi'$, (ii) ϕ' is just* true *or* false *or ϕ' mentions neither* true *nor* false*.*

It easy to see that it is possible to construct this fold operation such that it reduces the size of the formula ϕ at least by one if ϕ contains *true* or *false* but ϕ is not initially *true* or *false*.

3 Simplified Form

In this section, we first define *redundancy* and describe what it means for a formula to be in *simplified form*. We then highlight some important properties of simplified forms. Notions of redundancy similar to ours have been studied in other contexts, such as in *automatic test pattern generation* and *vacuity detection*; see Section 7 for a discussion.

Definition 4. ($\boldsymbol{\phi^+(\mathbf{L}), \phi^-(\mathbf{L})}$) *Let ϕ be a formula and let L be a leaf of ϕ. $\phi^+(L)$ is obtained by replacing L by* true *and applying the fold operation. Similarly, $\phi^-(L)$ is obtained by replacing L by* false *and folding the resulting formula.*

Example 1. Consider the formula:

$$\underbrace{x = y}_{L_0} \wedge (\underbrace{f(x) = 1}_{L_1} \vee (\underbrace{f(y) = 1}_{L_2} \wedge \underbrace{x + y \leq 1}_{L_3}))$$

Here, $\phi^+(L_1) = (x = y)$ and $\phi^-(L_2) = (x = y \wedge f(x) = 1)$.

Observe that for any formula ϕ, $\phi^+(L)$ is an overapproximation of ϕ, i.e., $\phi \Rightarrow \phi^+(L)$, and $\phi^-(L)$ is an underapproximation, i.e., $\phi^-(L) \Rightarrow \phi$. This follows immediately from Definition 4 and the monotonicity of NNF. Also, by construction, the sizes of $\phi^+(L)$ and $\phi^-(L)$ are at least one smaller than the size of ϕ.

Definition 5. (Redundancy) *We say a leaf L is* non-constraining *in formula ϕ if $\phi^+(L) \Rightarrow \phi$ and* non-relaxing *if $\phi \Rightarrow \phi^-(L)$. Leaf L is* redundant *if L is either non-constraining or non-relaxing.*

The following corollary follows immediately from definition:

Corollary 1. *If a leaf L is non-constraining, then $\phi \Leftrightarrow \phi^+(L)$, and if L is non-relaxing, then $\phi \Leftrightarrow \phi^-(L)$.*

Intuitively, if replacing a leaf L by *true* in formula ϕ results in an equivalent formula, then L does not constrain ϕ; hence, we call such a leaf non-constraining. A similar intuition applies for non-relaxing leaves.

Example 2. Consider the formula from Example 1. In this formula, leaves L_0 and L_1 are not redundant, but L_2 is redundant because it is non-relaxing. Leaf L_3 is both non-constraining and non-relaxing, and thus also redundant.

Note that if two leaves L_1 and L_2 are redundant in formula ϕ, this does not necessarily mean we can obtain an equivalent formula by replacing both L_1 and L_2 with *true* (if non-constraining) or *false* (if non-relaxing). This is the case because eliminating L_1 may render L_2 non-redundant and vice versa.

Definition 6. (Simplified Form) *We say a formula ϕ is in* simplified form *if no leaf mentioned in ϕ is redundant.*

Lemma 1. *If a formula ϕ is in simplified form, replacing any subset of the leaves used in ϕ by* true *or* false *does not result in an equivalent formula.*

Proof. The proof is by induction. If ϕ contains a single leaf, the property trivially holds. Suppose ϕ is of the form $\phi_1 \vee \phi_2$. Then, if ϕ has a simplification $\phi'_1 \vee \phi'_2$ where both ϕ'_1 and ϕ'_2 are simplified, then either $\phi'_1 \vee \phi_2$ or $\phi_1 \vee \phi'_2$ is also equivalent to ϕ. This is the case because $(\phi \Leftrightarrow \phi'_1 \vee \phi'_2) \wedge (\phi \not\Leftrightarrow \phi'_1 \vee \phi_2) \wedge (\phi \not\Leftrightarrow \phi_1 \vee \phi'_2)$ is unsatisfiable. An similar argument applies if the connective is $\wedge$. □

The following corollary follows directly from Lemma 1:

Corollary 2. *A formula ϕ in simplified form is satisfiable if and only if it is not syntactically* false *and valid if and only if it is syntactically* true*.*

This corollary is important in the context of on-line simplification in program analysis because, if formulas are kept in simplified form, then determining satisfiability and validity becomes just a syntactic check.

Observe that while a formula ϕ in simplified form is guaranteed not to contain redundancies, there may still exist a smaller formula ϕ' equivalent to ϕ. In particular, a non-redundant formula may be made smaller, for example, by factoring common subexpressions. We do not address this orthogonal problem in this paper, and the algorithm given in Section 4 does not change the structure of the formula.

Example 3. Consider the propositional formula $(a \wedge b) \vee (a \wedge c)$. This formula is in simplified form, but the equivalent formula $a \wedge (b \vee c)$ contains fewer leaves.

As this example illustrates, it is not possible to determine the equivalence of two formulas by checking whether their simplified forms are syntactically identical. Furthermore, as illustrated by the following example, the simplified form of a formula ϕ is not always guaranteed to be unique.

Example 4. Consider the formula $x = 1 \vee x = 2 \vee (1 \leq x \wedge x \leq 2)$ in the theory of linear integer arithmetic. The two formulas $x = 1 \vee x = 2$ and $1 \leq x \wedge x \leq 2$ are both simplified forms that can be obtained from the original formula.

Lemma 2. *If ϕ is a formula in simplified form, then $\neg\phi$ is also in simplified form after converting $\neg\phi$ to negation normal form.*

Proof. Suppose $\neg\phi$ was not in simplified form. Then, it would be possible to replace one leaf, say L, by *true* or *false* to obtain a strictly smaller, but equivalent formula. Now consider negating the simplified form of $\neg\phi$ to obtain ϕ' which is equivalent to ϕ. Note that the $\neg L$ is a leaf in ϕ, but not in ϕ'. Thus, ϕ could not have been in simplified form. □

Hence, if a formula is in simplified form, then its negation does not need to be resimplified, an important property for on-line simplification in program analysis. However, simplified forms are not preserved under conjunction or disjunction.

Lemma 3. *For every formula ϕ, there exists a formula ϕ' in simplified form such that (i) $\phi \Leftrightarrow \phi'$, and (ii)* $\text{size}(\phi') \leq \text{size}(\phi)$.

Proof. Consider computing ϕ' by checking every leaf L of ϕ for redundancy and replacing L by *true* if it is non-constraining and by *false* if it is non-relaxing. If this process is repeated until there are no redundant leaves, the resulting formula is in simplified form and contains at most as many leaves as ϕ. □

The above lemma states that converting a formula to its simplified form never increases the size of the formula. This property is desirable because, unlike other representations like BDDs that attempt to describe the formula compactly, computing a simplified form is guaranteed not to cause a worst-case blow-up. In the experience of the authors, this property is crucial in program verification.

4 Algorithm to Compute Simplified Forms

While the proof of Lemma 3 sketches a naive way of computing the simplified form of a formula ϕ, this approach is suboptimal because it requires repeatedly checking the satisfiability of a formula twice as large as ϕ until no more redundant leaves can be identified. In this section, we present a practical algorithm to compute simplified forms. For convenience, we assume formulas are represented as trees; however, the algorithm is easily modified to work on directed acyclic graphs, and in fact, our implementation uses DAGs to represent formulas. A node in the tree represents either an $\wedge$ or $\vee$ connective or a leaf. We assume connectives have at least two children but may have more than two.

4.1 Basic Algorithm

Recall that a leaf L is non-constraining if and only if $\phi^+(L) \Rightarrow \phi$ and non-relaxing if and only if $\phi \Rightarrow \phi^-(L)$. Since the size of $\phi^+(L)$ and $\phi^-(L)$ may be only one less than ϕ, checking whether L is non-constraining or non-relaxing using Definition 5 requires checking the validity of formulas twice as large as ϕ.

A key idea underlying our algorithm is that it is possible to check for redundancy of a leaf L by checking the validity of formulas no larger than ϕ. In particular, for each leaf L, our algorithm computes a formula $\alpha(L)$, called the *critical constraint* of L, such that (i) $\alpha(L)$ is no larger than ϕ, (ii) L is non-constraining if and only if $\alpha(L) \Rightarrow L$, and (iii) L is non-relaxing if and only if $\alpha(L) \Rightarrow \neg L$. This allows us to determine whether each leaf is redundant by determining the satisfiability of formulas no larger than the original formula ϕ.

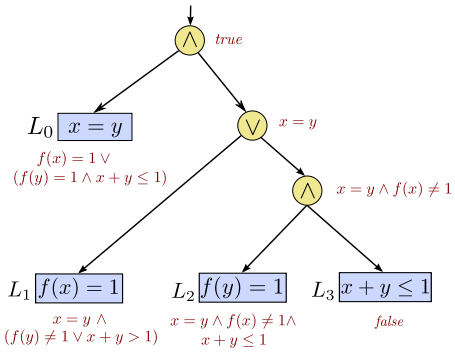


Fig. 1: The representation of the formula from Example 1. The critical constraint at each node is shown in red. Observe that the critical constraint for L_3 is *false*, making L_3 both non-constraining and non-relaxing. The critical constraint of L_2 implies its negation; hence, L_2 is non-relaxing.

Definition 7. (Critical constraint)

- *Let R be the root node of the tree. Then,* $\alpha(R) = \text{true}$.
- *Let N be any node other than the root node. Let P denote the parent of N in the tree, and let $S(N)$ denote the set of siblings of N. Let $\star$ denote $\neg$ if P is an $\vee$ connective, and nothing if P is an $\wedge$ connective. Then,*

$$\alpha(N) = \alpha(P) \wedge \bigwedge_{S_i \in S(N)} \star S_i$$

Intuitively, the critical constraint of a leaf L describes the condition under which L will be relevant for either permitting or disallowing a particular model of ϕ. Clearly, if the assignment to L is to determine whether ϕ is *true* or *false* for a given interpretation, then all the children of an $\wedge$ connective must be true if this $\wedge$ node is an ancestor of L; otherwise ϕ is already false regardless of the assignment to L. Also, observe that L is not relevant in permitting or disallowing a model of ϕ if some other path not involving L is satisfied because ϕ will already be true regardless of the truth value of L. Hence, the critical constraint includes the negation of the siblings at an $\vee$ connective while it includes the siblings themselves at an $\wedge$ node. The critical constraint can be viewed as a *context* in the general framework of *contextual rewriting* [12, 13]; see Section 7 for a discussion.

Example 5. Figure 1 shows the representation of the formula from Example 1 along with the critical constraints of each node.

Lemma 4. *A leaf L is non-constraining if and only if $\alpha(L) \Rightarrow L$.*

Proof. (Sketch) Suppose $\alpha(L) \Rightarrow L$, but L is constraining, i.e., the formula $\gamma = (\phi^+(L) \wedge \neg\phi)$ is satisfiable. Then, there must exist some model M of γ that satisfies $\phi^+(L)$ but not ϕ. For M to be a model of $\phi^+(L)$ but not ϕ, it must (i) assign all the children of any $\wedge$ node that is an ancestor of L to *true*, (ii) it must assign L to false, and (iii) it must assign any other children of an $\vee$ node that is an ancestor of L to *false*. By (i) and (iii), such a model must also satisfy $\alpha(L)$. Since $\alpha(L) \Rightarrow L$, M must also satisfy L, contradicting (ii). The other direction is analogous. □

Lemma 5. *A leaf L is non-relaxing if and only if $\alpha(L) \Rightarrow \neg L$.*

Proof. Similar to the proof of Lemma 4.

simplify(N, α)

- If N is a leaf:
  - If $\alpha \Rightarrow N$ return *true*
  - If $\alpha \Rightarrow \neg N$ return *false*
  - Otherwise return N
- If N is a connective, let C denote the ordered set of children of N, and let C' denote the new set of children of N .
  - For each $c_i \in C$:
    $$\begin{aligned} \alpha_i &= \alpha \ \wedge \ (\textstyle\bigwedge_{c_j \in C_{>i}} \star c_j) \ \wedge (\bigwedge_{c'_k \in C'_{<i}} \star c'_k) \\ c'_i &= \text{simplify}(c_i, \alpha_i) \\ C' &= C' \cup c'_i \end{aligned}$$
  - Repeat the previous step until $\forall i. c_i = c'_i$
  - If N is an $\wedge$ connective, return $\bigwedge_{c'_i \in C'} c'_i$
  - If N is an $\vee$ connective, return $\bigvee c'_i \in C' c'_i$

Fig. 2: The basic algorithm to reduce a formula N to its simplified form

We now formulate a simple recursive algorithm, presented in Figure 2, to reduce a formula ϕ to its simplified form. In this algorithm, N is a node representing the current subpart of the formula, and α denotes the critical constraint associated with N. If C is some ordered set, we use the notation $C_{<i}$ and $C_{>i}$ to denote the set of elements before and after index i in C respectively. Finally, we use the notation $\star$ as in Definition 7 to denote $\neg$ if the current node is an $\vee$ connective and nothing otherwise.

Observe that, in the algorithm of Figure 2, the critical constraint of each child c_i of a connective node is computed by using the new siblings c'_k that have been simplified. This is crucial for the correctness of the algorithm because, as pointed out in Section 3, if two leaves L_1 and L_2 are both initially redundant, it does not mean L_2 stays redundant after eliminating L_1 and vice versa. Using the simplified siblings in computing the critical constraint of c_i has the same effect as rechecking whether c_i remains redundant after simplifying sibling c_k.

Another important feature of the algorithm is that, at connective nodes, each child is simplified as long as any of their siblings change, i.e., the recursive invocation returns a new sibling not identical to the old one. The following example illustrates why this is necessary.

Example 6. Consider the following formula: $\underbrace{x \neq 1}_{L_1} \wedge \underbrace{(\underbrace{x \leq 0}_{L_2} \vee \underbrace{x > 2}_{L_3} \vee \underbrace{x = 1}_{L_4})}_{N}$

The simplified form of this formula is $x \leq 0 \vee x > 2$. Assuming we process child L_1 before N in the outer $\wedge$ connective, the critical constraint for L_1 is computed as $x \leq 0 \vee x > 2 \vee x = 1$, which implies neither L_1 nor $\neg L_1$. If we would not resimplify L_1 after simplifying N, the algorithm would (incorrectly) yield $x \neq 1 \wedge (x \leq 0 \vee x > 2)$ as the simplified form of the original formula. However, by resimplifying L_1 after obtaining a simplified $N' = (x \leq 0 \vee x > 2)$, we can now simplify the formula further because the new critical constraint of L_1, $(x \leq 0 \vee x > 2)$, implies $x \neq 1$.

Lemma 6. *The number of validity queries made in the algorithm of Figure 2 is bound by $2n^2$ where n denotes the number of leaves in the initial formula.*

Proof. First, observe that if any call to simplify yields a formula different from the input, the size of this formula must be at least one less than the original formula (see Lemma 3). Furthermore, the number of validity queries made in formula of size k without any simplifications is $2k$. Hence, the total number of validity queries is bound by $2n + 2(n-1) + \ldots + 2$ which is bound by $2n^2$. □

4.2 Making Simplification Practical

In the previous section, we showed that reducing a formula to its simplified form may require making a quadratic number of validity queries. However, these queries are not independent of one another in two important ways: First, all the formulas that correspond to validity queries share exactly the same set of leaves. Second, the simplification algorithm given in Figure 2 has a push-and-pop structure, which makes it possible to incrementalize queries. In the rest of this section, we discuss how we can make use of these observations to substantially reduce the cost of simplification in practice.

The first observation that all formulas whose satisfiability is queried during the algorithm share the same set of leaves has a fundamental importance when simplifying SMT formulas. Most modern SMT solvers use the DPLL($\mathcal{T}$) framework to solve formulas [14]. In the most basic version of this framework, leaves in a formula are treated as boolean variables, and this boolean overapproximation is then solved by a SAT solver. If the SAT solver generates a satisfying assignment that is not a valid assignment when theory-specific information is accounted for, the theory solver then produces (an ideally minimal) set of conflict clauses that is conjoined with the boolean overapproximation to prevent the SAT solver from generating at least this assignment in the future. Since the formulas solved by the SMT solver during the algorithm presented in Figure 2 share the same set of leaves, theory-specific conflict clauses can be gainfully reused. In practice, this means that after a small number of conflict clauses are learned, the problem of checking the validity of an SMT formula quickly converges to checking the satisfiability of a boolean formula.

The second important observation is that the construction of the critical constraint follows a push-pop stack structure. This is the case because the critical constraint from the parent node is reused, and additional constraints are pushed on the stack (i.e., added to the critical constraint) before the recursive call and (conceptually) popped from the stack after the recursive invocation. This stylized structure is important for making the algorithm practical because almost all modern SAT and SMT solvers support pushing and popping constraints to incrementalize solving. In addition, other tasks that often add overhead, such as CNF construction using Tseitin's encoding for the SAT solver, can also be incrementalized rather than done from scratch. In Section 6, we show the expected overhead of simplifying over solving grows sublinearly in the size of the formula in practice if the optimizations described in this section are used.

5 Integration with Program Analysis

We implemented the proposed algorithm in the Mistral constraint solver [15]. To tightly integrate simplification into a program analysis system, we design the

interface of Mistral such that instead of giving a "yes/no" answer to satisfiability and validity queries, it yields a formula ϕ' in simplified form. Recall that ϕ is satisfiable (valid) if and only if ϕ' is not syntactically *false* (*true*); hence, in addition to obtaining a simplified formula, the program analysis system can check whether the formula is satisfiable by syntactically checking if ϕ' is not *false*. After a satisfiability query is made, we then replace all instances of ϕ with ϕ' such that future formulas that would be constructed by using ϕ are instead constructed using ϕ'. This functionality is implemented efficiently through a shared constraint representation. Hence, Mistral's interface is designed to be useful for program analysis systems that incrementally construct formulas from existing formulas and make many intermediary satisfiability or validity queries. Examples of such systems include, but are not limited to, [10, 16, 7–9, 17].

6 Experimental Results

In this section, we report on our experience using on-line simplification in the context of program analysis. Since the premise of this work is that simplification is useful only if applied continuously during the analysis, we do not evaluate the proposed algorithm on solving off-line benchmarks such as the SMT-LIB. *In particular, the proposed technique is not meant as a preprocessing step before solving and is not expected to improve solving time on individual constraints.*

6.1 Impact of On-line Simplification on Analysis Scalability

In our first experiment, we integrate Mistral into the Compass program verification system. Compass [16] is a path- and context-sensitive program analysis system for analyzing C programs, integrating reasoning about both arrays and contents of the heap. Compass checks memory safety properties, such as buffer overruns, null dereferences, casting errors, and uninitialized memory; it can also check user-provided assertions. Compass generates constraints in the combined theory of uninterpreted functions and linear integer arithmetic, and as typical of many program analysis systems [18, 10, 16, 17], constraints generated by Compass become highly redundant over time, as new constraints are obtained by combining existing constraints. Most importantly, unlike other systems that employ various (usually incomplete) heuristics to control formula size, Compass tracks program conditions precisely without identifying a relevant set of predicates to track. Hence, this experiment is used to illustrate that a program analysis system can be made scalable through on-line simplification instead of using specialized heuristics, such as the ones discussed in Section 1, to control formula size.

In this experiment, we run Compass on 811 program analysis benchmarks, totalling over 173,000 lines of code, ranging from small programs with 20 lines to real-world applications, such as OpenSSH, with over 26,000 lines. For each benchmark, we fix a time-out of 3600 seconds and a maximum memory of 4 GB. Any run exceeding either limit was aborted and assumed to take 3600 seconds.

Figure 3 compares Compass's running times on these benchmarks with and without on-line simplification. The x-axis shows the number of lines of code for various benchmarks and the y-axis shows the running time in seconds. Observe that both axes are log scale. The blue (dotted) line shows the performance of

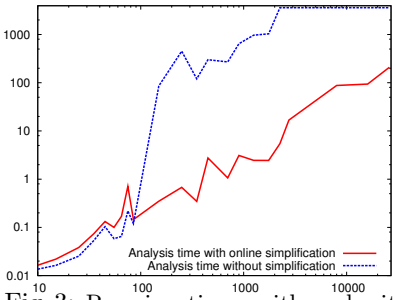
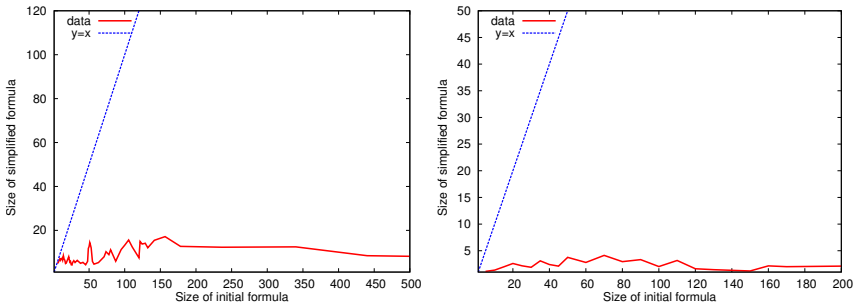


Fig. 3: Running times with and without simplification

Compass without on-line simplification while the red (solid) line shows the performance of Compass using the simplification algorithm presented in this paper and using the improvements from Section 4.2. In the setting that does not use on-line simplification, Mistral returns the formula unchanged if it is satisfiable and *false* otherwise. As this figure shows, Compass performs dramatically better with on-line simplification on any benchmark exceeding 100 lines. For example, on benchmarks with an average size of 1000 lines, Compass performs about two orders of magnitude better with on-line simplification, and can analyze programs of this size in just a few seconds. Furthermore, using on-line simplification, Compass can analyze benchmarks with a few ten thousand lines of code, such as OpenSSH, in the order of just a few minutes without employing any heuristics to identify relevant conditions.

6.2 Redundancy in Program Analysis Constraints

This dramatic impact of simplification on scalability is best understood by considering how redundant formulas become when on-line simplification is disabled when analyzing the same set of 811 program analysis benchmarks. Figure 4(a) plots the size of the initial formula vs. the size of the simplified formula when formulas generated by Compass are not continuously simplified. The $x = y$ line is plotted as a comparison to show the worst-case when the simplified formula is no smaller than the original formula. As this figure shows, while formula sizes grow very quickly without on-line simplification, these formulas are very redundant, and much smaller formulas are obtained by simplifying them. We would like to point out that the redundancies present in these formulas cannot be detected through simple syntactic checks because Mistral still performs extensive syntactic simplifications, such as detecting duplicates, syntactic contradictions and tautologies, and folding constants.



(a) Size of initial formula vs. size of simplified formula in Compass without simplification

(b) Size of initial formula vs. size of simplified formula in Saturn

Fig. 4: Reduction in the Size of Formulas

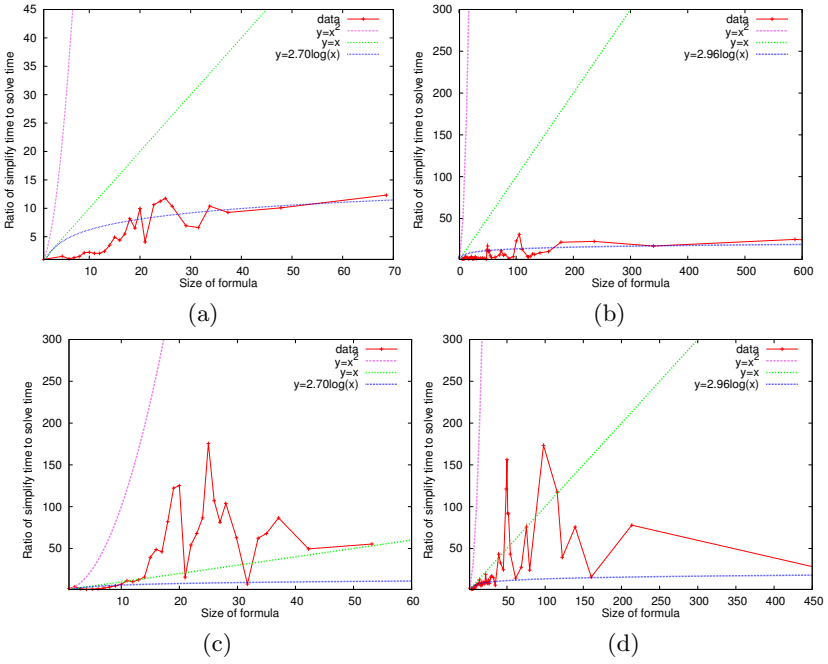


Fig. 5: Complexity of Simplification in Practice

To demonstrate that Compass is not the only program analysis system that generates redundant constraints, we also plot in Figure 4(b) the original formula size vs. simplified formula size on constraints obtained on the same benchmarks by the Saturn program analysis system [10]. First, observe that the constraints generated by Saturn are also extremely redundant. In fact, their average size after simplification is 1.93 whereas the average size before simplification is 73. Second, observe that the average size of simplified constraints obtained from Saturn is smaller than the average simplified formula size obtained from Compass. This difference is explained by two factors: (i) Saturn is significantly less precise than Compass, and (ii) it adopts heuristics to control formula size.

The reader may not find it surprising that the redundant formulas generated by Compass can be dramatically simplified. That is, of course, precisely the point. Compass gains both better precision and simpler engineering from constructing straightforward formulas and then simplifying them because it does not need to heuristically decide in advance which predicates are important. But these experiments also show that the formulas generated by Compass are not unusually redundant to begin with: As the Saturn experiment shows, because analysis systems build formulas compositionally guided by the structure of the program, even highly-engineered systems like Saturn, designed without the assumption of pervasive simplification, can construct very redundant formulas.

6.3 Complexity of Simplification in Practice

In another set of experiments, we evaluate the performance of our simplification algorithm on over 93,000 formulas obtained from our 811 program analysis

benchmarks. Recall from Lemma 6 that simplification may require a quadratic number of validity checks. Since the size of the formulas whose validity is checked by the algorithm is at most as large as the original formula, the ratio of simplifying to solving could, in the worst case, be quadratic in the size of the original formula. Fortunately, with the improvements discussed in Section 4.2, we show empirically that simplification adds sub-linear overhead over solving in practice.

Figure 5 shows a detailed evaluation of the performance of the simplification algorithm. The data used in graphs 5a and 5c is obtained from analysis runs where simplification is performed, while graphs 5b and 5d are from experiments where no simplification is performed. We include the data from runs where no simplification is performed to demonstrate that the simplification algorithm also performs well on larger constraints with several hundred leaves. In all of these graphs, the red line marks data points, the blue line marks the function best fitting the data, the green line marks $y = x$, and the pink line marks $y = x^2$. The top two graphs are obtained from runs that employ the improvements described in Section 4.2 whereas the two bottom graphs are obtained from runs that do not. Observe that in graphs 5a and 5b, the average ratio of simplification to solve time seems to grow sublinearly in formula size. In fact, from among the family of formulas $y = cx^2$, $y = cx$, and $y = c \cdot log(x)$, the data in figures 4a and 4b are best approximated by $y = 2.70 \cdot log(x)$ and $y = 2.96 \cdot log(x)$ with asymptotic standard errors 1.98% and 2.42% respectively. On the other hand, runs that do not exploit the dependence between different implication queries exhibit much worse performance, often exceeding the $y = x$ line. These experiments show the importance of exploiting the interdependence between different implication queries and validate our hypothesis that simplifying SMT formulas converges quickly to simplifying SAT formulas when queries are incrementalized. These experiments also show that the overhead of simplifying vs. solving can be made manageable since the ratio of simplifying to solving seems to grow very slowly in the size of the formula.

7 Related Work

Finding simpler representations of boolean circuits is a well-studied problem in logic synthesis and automatic test pattern generation (ATPG) [19–21]. Our definition of redundancy is reminiscent of the concept of *undetectable faults* in circuits, where pulling an input to 0 (false) or 1 (true) is used to identify redundant circuitry. However, in contrast to the definition of size considered in this paper, ATPG and logic synthesis techniques are concerned with minimizing DAG size, representing the size of the circuit implementing a formula. As a result, the notion of redundancy considered in this paper is different from the notion of redundancy addressed by these techniques. In particular, in our setting, one subpart of the formula may be redundant while another syntactically identical subpart may not. In this paper, we consider different definitions of size and redundancy because except for a few operations like substitution, most operations performed on constraints in a program analysis system are sensitive to the "tree size" of the formula, although these formulas are represented as DAGs internally. Therefore, formulas we consider do not exhibit reconvergent fanout and every

leaf has exactly one path from the root of the formula. This observation makes it possible to formulate an algorithm based on critical constraints for simplifying formulas in an arbitrary theory. Furthermore, we apply this simplification technique to on-line constraint simplification in program analysis.

The algorithm we present for converting formulas to simplified form can be understood as an instance of a *contextual rewrite system* [12, 13]. In contextual rewriting systems, if a precondition, called a *context*, is satisfied, a rewrite rule may be applied. In our algorithm, the critical constraint can be seen as a context that triggers a rewrite rule $L \rightarrow \mathit{true}$ if L is implied by the critical constraint α, and $L \rightarrow \mathit{false}$ if α implies $\neg L$. While contextual rewriting systems have been used for simplifying constraints within the solver [13], our goal is to generate an *equivalent* (rather than equisatisfiable) formula that is in simplified form. Furthermore, we propose simplification as an alternative to heuristic-based predicate selection techniques used for improving scalability of program analysis systems.

Finding redundancies in formulas has also been studied in the form of *vacuity detection* in temporal logic formulas [22, 23]. Here, the goal is to identify vacuously valid subparts of formulas, indicating, for example, a specification error. In contrast, our focus is giving a practical algorithm for on-line simplification of program analysis constraints.

The problem of representing formulas compactly has received attention from many different angles. For example, BDDs attempt to represent propositional formulas concisely, but they suffer from the variable ordering problem and are prone to a worst-case exponential blow-up [24]. BDDs have also been extended to other theories, such as linear arithmetic [25, 26]. In contrast to these approaches, a formula in simplified form is never larger than the original formula. Loveland and Shostak address the problem of finding a minimal representation of formulas in normal form [27]; in contrast, our approach does not require formulas to be converted to DNF or CNF.

Various rewrite-based simplification rules have also been successfully applied as a preprocessing step for solving, usually for bit-vector arithmetic [28, 29]. These rewrite rules are syntactic and theory-specific; furthermore, they typically yield equisatisfiable rather than equivalent formulas and give no goodness guarantees. In contrast, the technique described in this paper is not meant as a preprocessing step for solving and guarantees non-redundancy.

The importance of on-line simplification of program analysis constraints has been studied previously in the very different setting of set constraints [18]. Simplification based on syntactic rewrite-rules has also been shown to improve the performance of a program analysis system significantly in [30].

References

1. Een, N., Sorensson, N.: MiniSat: A SAT solver with conflict-clause minimization. 8th SAT (2005)
2. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. TACAS (2008) 337–340
3. Dutertre, B., De Moura, L.: The yices smt solver. Technical report, SRI (2006)
4. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: CAV, Springer-Verlag (2008) 299–303

5. Barrett, C., Tinelli, C.: CVC3. In: CAV. Volume 4590 of Lecture Notes in Computer Science., Springer-Verlag (July 2007) 298–302 Berlin, Germany.
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. JACM **50**(5) (2003) 752–794
7. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, ACM New York, NY, USA (2002) 58–70
8. Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: POPL, NY, USA (2002) 1–3
9. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. ACM SIGPLAN Notices **37**(5) (2002) 57–68
10. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: POPL. Volume 40., ACM New York, NY, USA (2005) 351–363
11. Bugrara, S., Aiken, A.: Verifying the safety of user pointer dereferences. In: IEEE Symposium on Security and Privacy, 2008. SP 2008. (2008) 325–338
12. Lucas, S.: Fundamentals of Contex-Sensitive Rewriting. Lecture Notes in Computer Science (1995) 405–412
13. Armando, A., Ranise, S.: Constraint contextual rewriting. Journal of Symbolic Computation **36**(1) (2003) 193–216
14. Tinelli, C.: A DPLL-based calculus for ground satisfiability modulo theories. Lecture notes in computer science (2002) 308–319
15. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: CAV, Springer (2009)
16. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: To appear in ESOP. (2010)
17. Babić, D., Hu, A.J.: Calysto: Scalable and Precise Extended Static Checking. In: ICSE, New York, NY, USA, ACM (May 2008) 211–220
18. Faehndrich, M., Foster, J., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: PLDI, ACM (1998) 96
19. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In: DAC. (2006) 532–535
20. Mishchenko, A., Brayton, R., Jiang, J., Jang, S.: SAT-based logic optimization and resynthesis. Proc. IWLS'07 358–364
21. Kim, J., Silva, J., Savoj, H., Sakallah, K.: RID-GRASP: Redundancy identification and removal using GRASP. In: International Workshop on Logic Synthesis. (1997)
22. Kupferman, O., Vardi, M.: Vacuity detection in temporal model checking. International Journal on Software Tools for Technology Transfer **4**(2) (2003) 224–233
23. Armoni, R., Fix, L., Flaisher, A., Grumberg, O., Piterman, N., Tiemeyer, A., Vardi, M.: Enhanced vacuity detection in linear temporal logic. LNCS (2003) 368–380
24. Bryant, R.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys (CSUR) **24**(3) (1992) 293–318
25. Bryant, R., Chen, Y.: Verification of arithmetic functions with BMDs (1994)
26. Clarke, E., Fujita, M., Zhao, X.: Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In: ICCAD. (1995)
27. Loveland, D., Shostak, R.: Simplifying interpreted formulas. In: Proc. 5th Conf. on Automated Deduction (CADE). Volume 87., Springer (1987) 97–109
28. Ganesh, V., Dill, D.: A decision procedure for bit-vectors and arrays. Lecture Notes in Computer Science **4590** (2007) 519
29. Jha, S., Limaye, R., Seshia, S.: Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In: In CAV Lecture Notes in Comp. Sc., Springer (2009)
30. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. SIGPLAN Not. **44**(6) (2009) 363–374