Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers (extended version)

Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers

Alexandra Bugariu, Arshavir Ter-Gabrielyan, and Peter Müller

Department of Computer Science, ETH Zurich, Switzerland
{alexandra.bugariu,ter-gabrielyan,peter.mueller}@inf.ethz.ch

Abstract. Universal quantifiers occur frequently in proof obligations produced by program verifiers, for instance, to axiomatize uninterpreted functions and to express properties of arrays. SMT-based verifiers typically reason about them via E-matching, an SMT algorithm that requires syntactic matching patterns to guide the quantifier instantiations. Devising good matching patterns is challenging. In particular, overly restrictive patterns may lead to spurious verification errors if the quantifiers needed for a proof are not instantiated; they may also conceal unsoundness caused by inconsistent axiomatizations. In this paper, we present the first technique that identifies and helps the users remedy the effects of overly restrictive matching patterns. We designed a novel algorithm to synthesize missing triggering terms required to complete a proof. Tool developers can use this information to refine their matching patterns and prevent similar verification errors, or to fix a detected unsoundness.

1 Introduction

Proof obligations frequently contain universal quantifiers, both in the specification and to encode the semantics of the programming language. Most program verifiers [20,38,11,14,5,8,4] rely on SMT solvers to discharge the proof obligations via E-matching [13]. This SMT algorithm requires syntactic matching patterns of ground terms (called *patterns* in the following), to control the instantiations. The pattern $\{\mathtt{f}(x, y)\}$ in the formula $\forall x\colon \mathrm{Int}, y\colon \mathrm{Int} :: \{\mathtt{f}(x, y)\}\ (x = y) \wedge \neg \mathtt{f}(x, y)$ instructs the solver to instantiate the quantifier *only* when it finds a *triggering term* that matches the pattern, e.g., $\mathtt{f}(7, z)$. The patterns can be written manually or inferred automatically. However, devising them is challenging [21,24]. Too permissive patterns may lead to unnecessary instantiations that slow down verification or even cause non-termination (if each instantiation produces a new triggering term, in a so-called matching loop [13]). Overly restrictive patterns may prevent the instantiations needed to complete a proof; they cause two major problems in program verification, incompleteness and undetected unsoundness.

Incompleteness. Overly restrictive patterns may cause spurious verification errors when the proof of *valid* proof obligations fails. Fig. 1 illustrates this. The integer `x` represents the address of a node, and the uninterpreted functions `len` and `nxt` encode operations on linked lists. The axiom defines `len`: its result is

```
function len(x: int): int;
function nxt(x: int): int;

axiom (forall x: int :: {len(nxt(x))}
        len(x) > 0 &&
        (nxt(x) != x ==> len(x) == len(nxt(x)) + 1) &&
        (nxt(x) == x ==> len(x) == 1));

procedure trivial()
{
  assert len(7) > 0;
}
```

Fig. 1. Example (in Boogie [7]) that leads to a spurious error. The assertion follows from the axiom, but the axiom does not get instantiated without a triggering term.

positive and the last node points to itself. The assertion directly follows from the axiom, but the proof fails because the proof obligation does not contain the triggering term `len(nxt(7))`; thus, the axiom does not get instantiated. Realistic proof obligations often contain hundreds of quantifiers, which makes the manual identification of missing triggering terms extremely difficult.

Unsoundness. Most of the universal quantifiers in proof obligations appear in axioms over uninterpreted functions (to encode type information, heap models, datatypes, etc.). To obtain sound results, these axioms must be consistent (i.e., satisfiable); otherwise all proof obligations hold trivially. Consistency can be proved once and for all by showing the existence of a model, as part of the soundness proof. However, this solution is difficult to apply for practical verifiers, which generate axioms *dynamically*, depending on the program to be verified. Proving consistency then requires verifying the algorithm that generates the axioms for all possible inputs, and needs to consider many subtle issues [12,31,22].

A more practical approach is to check if the axioms generated for a given program are consistent. However, this check also depends on triggering: an SMT solver may fail to prove unsat if the triggering terms needed to instantiate the contradictory axioms are missing. The unsoundness can thus remain undetected.

For example, Dafny's [20] sequence axiomatization from June 2008 contained an inconsistency found only over a year later. A fragment of this axiomatization is shown in Fig. 2. It expresses that empty sequences and sequences obtained through the `Build` operation are well-typed (F_0–F_2), that the length of a type-correct sequence must be non-negative (F_3), and that `Build` constructs a new sequence of the required length (F_4). The intended behavior of `Build` is to update the element at index i_4 in sequence s_4 to v_4. However, since there are no constraints on the parameter len_4, `Build` can be used with a negative length, leading to a contradiction with F_3. This error cannot be detected by checking the satisfiability of the formula $F_0 \wedge \ldots \wedge F_4$, as no axiom gets instantiated.

$$F_0 : \forall t_0 : \mathrm{V} :: \{\texttt{Type}(t_0)\}\ t_0 = \texttt{ElemType}(\texttt{Type}(t_0))$$
$$F_1 : \forall t_1 : \mathrm{V} :: \{\texttt{Empty}(t_1)\}\ \texttt{typ}(\texttt{Empty}(t_1)) = \texttt{Type}(t_1)$$
$$F_2 : \forall s_2 : \mathrm{U}, i_2 : \mathrm{Int}, v_2 : \mathrm{U}, len_2 : \mathrm{Int} :: \{\texttt{Build}(s_2, i_2, v_2, len_2)\}$$
$$\texttt{typ}(\texttt{Build}(s_2, i_2, v_2, len_2)) = \texttt{Type}(\texttt{typ}(v_2))$$
$$F_3 : \forall s_3 : \mathrm{U} :: \{\texttt{Length}(s_3)\}$$
$$\neg(\texttt{typ}(s_3) = \texttt{Type}(\texttt{ElemType}(\texttt{typ}(s_3))) \vee (0 \leq \texttt{Length}(s_3))$$
$$F_4 : \forall s_4 : \mathrm{U}, i_4 : \mathrm{Int}, v_4 : \mathrm{U}, len_4 : \mathrm{Int} :: \{\texttt{Length}(\texttt{Build}(s_4, i_4, v_4, len_4))\}$$
$$\neg(\texttt{typ}(s_4) = \texttt{Type}(\texttt{typ}(v_4))) \vee (\texttt{Length}(\texttt{Build}(s_4, i_4, v_4, len_4)) = len_4)$$

Fig. 2. Fragment of an old version of Dafny's sequence axiomatization. U and V are uninterpreted types. All the named functions are uninterpreted. To improve readability, we use mathematical notation throughout this paper instead of SMT-LIB syntax [10].

This work. Both discharging a proof obligation and revealing an inconsistency in an axiomatization require an SMT solver to prove unsat via E-matching. Given an SMT formula for which E-matching yields *unknown* due to insufficient quantifier instantiations, our technique generates suitable triggering terms that allow the solver to complete the proof. These terms enable users to understand and remedy the revealed completeness or soundness issue. Since the SMT queries for the verification of different input programs are typically very similar, fixing such issues benefits the verification of many or even all future runs of the verifier.

Fixing the incompleteness. For Fig. 1, our technique finds the triggering term `len(nxt(7))`, which allows one to fix the incompleteness. Tool *users* (who cannot change the axioms) can add the term to the program; e.g., adding `var t: int; t := len(nxt(7))` before the assertion has no effect on the execution, but triggers the instantiation of the axiom. Tool *developers* can devise less restrictive patterns. For instance, they can move the conjunct `len(x) > 0` to a separate axiom with the pattern `{len(x)}` (simply changing the axiom's pattern to `{len(x)}` would cause matching loops). Alternatively, tool developers can adapt the encoding to emit additional triggering terms enforcing certain instantiations [18,21].

Fixing the unsoundness. For Fig. 2, our triggering term $\texttt{Length}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))$ (for a fresh value v) is sufficient to detect the unsoundness (see Appx. A). Tool developers can use this information to add a precondition to F_4, which prevents the construction of sequences with negative lengths.

Soundness modulo patterns. Fig. 3 illustrates a different scenario: Boogie's [7] map axiomatization is inconsistent, but Boogie does not produce any terms that could trigger quantifier instantiations to reveal it. The root cause of the problem is F_2, which states that storing a key-value pair into a map results in a new map with a (potentially) *different* type. Our technique synthesizes triggering terms

that instantiate the axioms and allow the solver to prove that two different types (such as Boolean and Int) are equal, which is equivalent to false.

As this inconsistency cannot be triggered by Boogie, it does not affect Boogie's soundness. It is nevertheless important to detect it because it could surface if Boogie was extended to support quantifier instantiation algorithms that are not based on E-matching (such as MBQI [16]) or first-order provers. This example also shows that the problems tackled in this paper cannot be solved by simply switching to other instantiation strategies: these are not the preferred choices of most verifiers [20,38,11,14,5,8,4], and they might produce unsound results for verifiers that were designed to use E-matching.

$$
\begin{aligned}
F_0 &: \forall kt_0 : \mathrm{V}, vt_0 : \mathrm{V} :: \{\mathtt{Type}(kt_0, vt_0)\}\ \mathtt{ValueTypeInv}(\mathtt{Type}(kt_0, vt_0)) = vt_0 \\
F_1 &: \forall m_1 : \mathrm{U}, k_1 : \mathrm{U}, v_1 : \mathrm{U} :: \{\mathtt{Select}(m_1, k_1, v_1)\} \\
&\qquad \mathtt{typ}(\mathtt{Select}(m_1, k_1, v_1)) = \mathtt{ValueTypeInv}(\mathtt{typ}(m_1)) \\
F_2 &: \forall m_2 : \mathrm{U}, k_2 : \mathrm{U}, x_2 : \mathrm{U}, v_2 : \mathrm{U} :: \{\mathtt{Store}(m_2, k_2, x_2, v_2)\} \\
&\qquad \mathtt{typ}(\mathtt{Store}(m_2, k_2, x_2, v_2)) = \mathtt{Type}(\mathtt{typ}(k_2), \mathtt{typ}(v_2)) \\
F_3 &: \forall m_3 : \mathrm{U}, k_3 : \mathrm{U}, x_3 : \mathrm{U}, v_3 : \mathrm{U}, k_3' : \mathrm{U}, v_3' : \mathrm{U} :: \{\mathtt{Select}(\mathtt{Store}(m_3, k_3, x_3, v_3), k_3', v_3')\} \\
&\qquad (k_3 = k_3') \vee (\mathtt{Select}(\mathtt{Store}(m_3, k_3, x_3, v_3), k_3', v_3') = \mathtt{Select}(m_3, k_3', v_3'))
\end{aligned}
$$

Fig. 3. Fragment of Boogie's map axiomatization, which is inconsistent at the SMT level. U and V are uninterpreted types. All the named functions are uninterpreted.

Contributions. This paper makes the following technical contributions:

1. We present the first automated technique that allows developers to detect *completeness* issues in program verifiers and *soundness* problems in their axiomatizations, and helps them to devise better triggering strategies for *all* future runs of their tool.
2. We developed a novel algorithm for synthesizing the triggering terms necessary to complete unsatisfiability proofs using E-matching. Since quantifier instantiation is undecidable for first-order formulas over uninterpreted functions, our algorithm might not terminate. However, all identified triggering terms are indeed sufficient to complete the proof; there are no false positives.
3. We evaluated our technique on benchmarks with known triggering problems from four program verifiers. Our experimental results show that it successfully synthesized the missing triggering terms in 65,62% of the cases, and can significantly reduce the human effort in localizing and fixing the errors.

Outline. The rest of the paper is organized as follows: Sec. 2 gives an overview of our technique; the details follow in Sec. 3. In Sec. 4, we present our experimental results. We discuss related work in Sec. 5, and conclude in Sec. 6.

2 Overview

Our goal is to synthesize missing triggering terms, i.e., concrete instantiations for (a small subset of) the quantified variables of an SMT formula, which are necessary for the solver to prove its unsatisfiablity. Intuitively, these triggering terms include *counter-examples* to the satisfiability of the formula and can be obtained from a model of its negation. $\mathtt{I} = \forall n\colon \mathrm{Int} :: n > 7$ is unsatisfiable, and a counter-example $n = 6$ is a model of its negation $\neg\mathtt{I} = \exists n\colon \mathrm{Int} :: n \leq 7$.

However, this idea does not directly apply to formulas over uninterpreted functions, which are common in proof obligations. The negation of $\mathtt{I} = \forall n\colon \mathrm{Int} :: \mathtt{f}(n, 7)$, where $\mathtt{f}$ is an uninterpreted function, is $\neg\mathtt{I} = \forall \mathtt{f}, \exists n\colon \mathrm{Int} :: \neg\mathtt{f}(n, 7)$. This is a second-order constraint (it quantifies over functions), and cannot be encoded in SMT, which supports only first-order logic. Therefore, we take a different approach.

Let F be an arbitrary formula. We define its *approximation* as:

$$F_{\approx} = F[\exists\overline{\mathtt{f}} \,/\, \forall\overline{\mathtt{f}}] \qquad (*)$$

where $\overline{\mathtt{f}}$ are uninterpreted functions. The approximation thus considers only *one* interpretation, not *all* possible interpretations for each uninterpreted function.

We construct a *candidate* triggering term from a model of $\neg\mathtt{I}_{\approx}$ and check if it is sufficient to prove that $\mathtt{I}$ is unsatisfiable (due to the approximation, a model is no longer guaranteed to be a counter-example for the original formula).

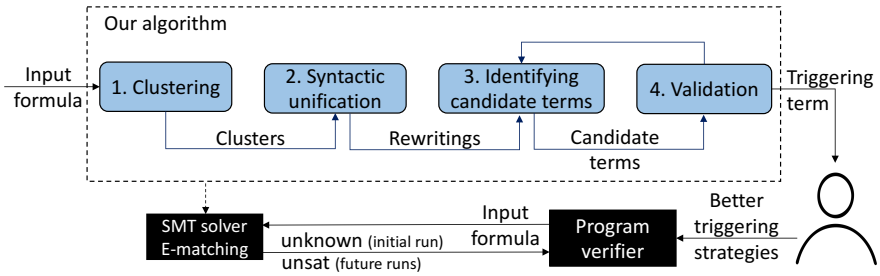

Fig. 4. Main steps of our algorithm, which helps the developers of program verifiers devise better triggering strategies for this and future runs of the verifier with E-matching. Rounded boxes depict processing steps and arrows data.

The four main steps of our algorithm are depicted in Fig. 4. The algorithm is stand-alone, i.e., not integrated into, nor dependent on any specific SMT solver. We illustrate it on the inconsistent axioms from Fig. 5 (which we assume are part of a larger axiomatization). To show that $\mathtt{I} = F_0 \wedge F_1 \wedge \ldots$ is unsatisfiable, the solver requires the triggering term $\mathtt{f}(\mathtt{g}(7))$. The corresponding instantiations of F_0 and F_1 generate contradictory constraints: $\mathtt{f}(\mathtt{g}(7)) \neq 7$ and $\mathtt{f}(\mathtt{g}(7)) = 7$. In the following, we explain how we obtain this triggering term systematically.

$$F_0\colon \quad \forall x_0 : \text{Int} :: \{\mathtt{f}(x_0)\} \quad \mathtt{f}(x_0) \neq 7$$
$$F_1\colon \quad \forall x_1 : \text{Int} :: \{\mathtt{f}(\mathtt{g}(x_1))\} \quad \mathtt{f}(\mathtt{g}(x_1)) = x_1$$

Fig. 5. Formulas that set contradictory constraints on the function `f`. Synthesizing the triggering term $\mathtt{f}(\mathtt{g}(7))$ requires theory reasoning and syntactic term unification.

Step 1: Clustering. As typical proof obligations or axiomatizations contain hundreds of quantifiers, exploring combinations of triggering terms for all of them does not scale. To prune the search space, we exploit the fact that `I` is unsatisfiable only if there exist instantiations of some (in the worst case all) of the *quantified* conjuncts F of `I` such that they produce contradictory constraints on some uninterpreted functions. (If there is a contradiction among the quantifier-free conjuncts, the solver will detect it without our technique.)

We thus identify *clusters* C of formulas F that share function symbols and then process each cluster separately. In Fig. 5, F_0 and F_1 share the function symbol `f`, so we build the cluster $C = F_0 \wedge F_1$.

Step 2: Syntactic unification. The formulas within clusters usually contain uninterpreted functions applied to *different* arguments (e.g., `f` is applied to x_0 in F_0 and to $\mathtt{g}(x_1)$ in F_1). We thus perform syntactic unification to identify *sharing constraints* on the quantified variables (which we call *rewritings* and denote their set by R) such that instantiations that satisfy these rewritings generate formulas with common terms (on which they might set contradictory constraints). F_0 and F_1 share the term $\mathtt{f}(\mathtt{g}(x_1))$ if we perform the rewritings $R = \{x_0 = \mathtt{g}(x_1)\}$.

Step 3: Identifying candidate triggering terms. The cluster $C = F_0 \wedge F_1$ from step 1 contains a contradiction if there exists a formula F in C such that: (1) F is unsatisfiable by itself, or (2) F contradicts at least one of the remaining formulas from the cluster.

To address scenario (1), we ask an SMT solver for a model of the formula $G = \neg C_{\approx}$, where $\neg C_{\approx}$ is defined in (*) above. After Skolemization, G is quantifier-free, so the solver is generally able to provide a model if one exists. We then obtain a candidate triggering term by substituting the quantified variables from the patterns of the formulas in C with their corresponding values from the model.

However, scenario (1) is not sufficient to expose the contradiction from Fig. 5, since both F_0 and F_1 are individually satisfiable. Our algorithm thus also derives *stronger* G formulas corresponding to scenario (2). That is, it will next consider the case where F_0 contradicts F_1, whose encoding into first-order logic is: $\neg F_{0\approx} \wedge F_1 \wedge \bigwedge R$, where R is the set of rewritings identified in step 2, used to connect the quantified variables. This formula is universally-quantified (since F_1 is), so the solver cannot prove its satisfiability and generate models. We solve this problem by requiring F_0 to contradict the *instantiation* of F_1, which is a weaker constraint. Let F be an arbitrary formula. We define its *instantiation* as:

$$F_{Inst} = F[\exists \overline{\mathtt{x}} \,/\, \forall \overline{\mathtt{x}}] \qquad (**)$$

Then $G = \neg F_{0\approx} \wedge F_{1\,Inst} \wedge \bigwedge R$ is $(\mathtt{f}(x_0) = 7) \wedge (\mathtt{f}(\mathtt{g}(x_1)) = x_1) \wedge (x_0 = \mathtt{g}(x_1))$. (To simplify the notation, here and in the following formulas, we omit existential quantifiers.) All its models set x_1 to 7. Substituting x_0 by $\mathtt{g}(x_1)$ (according to R) and x_1 by 7 (its value from the model) in the patterns of F_0 and F_1 yields the candidate triggering term $\mathtt{f}(\mathtt{g}(7))$.

Step 4: Validation. Once we have found a candidate triggering term, we add it to the original formula `I` (wrapped in a fresh uninterpreted function, to make it available to E-matching, but not affect the input's satisfiability) and check if the solver can prove unsat. If so, our algorithm terminates successfully and reports the synthesized triggering term (after a minimization step that removes unnecessary sub-terms); otherwise, we go back to step 3 to obtain another candidate. In our example, the triggering term $\mathtt{f}(\mathtt{g}(7))$ is sufficient to complete the proof.

3 Synthesizing Triggering Terms

Next, we define the input formulas (Sec. 3.1) and explain the details of our algorithm (Sec. 3.2). Its extensions and limitations follow in Appx. C and Appx. F.

3.1 Input formula

To simplify our algorithm, we pre-process the inputs (i.e., the proof obligations or the axioms of a verifier): we Skolemize existential quantifiers and transform all propositional formulas into *negation normal form* (NNF), where negation is applied only to literals and the only logical connectives are conjunction and disjunction; we also apply the distributivity of disjunction over conjunction and split conjunctions into separate formulas. These steps preserve satisfiability and the semantics of patterns (Appx. E discusses potential scalability issues). The resulting formulas follow the grammar in Fig. 6. Literals L may include interpreted and uninterpreted functions, variables and constants. Free variables are nullary functions. Quantified variables can have interpreted or uninterpreted types, and the pre-processing ensures that their names are globally unique. We assume that each quantifier is equipped with a pattern P (if none is provided, we run the solver to infer one). Patterns are combinations of *uninterpreted* functions and must mention *all* quantified variables. Since there are no existential quantifiers after Skolemization, we use the term *quantifier* to denote *universal quantifiers*.

$$\mathtt{I} ::= F\ (\wedge F)^* \qquad B ::= D\ (\vee D)^*$$
$$F ::= B \mid \forall \overline{x} :: \{P(\overline{x})\}\ B \qquad D ::= L \mid \neg L \mid \forall \overline{x} :: \{P(\overline{x})\}\ F$$

Fig. 6. Grammar of input formulas `I`. Inputs are conjunctions of formulas F, which are (typically quantified) disjunctions of literals (L or $\neg L$) or nested quantified formulas. Each quantifier is equipped with a pattern P. $\overline{x}$ denotes a (non-empty) list of variables.

Algorithm 1: Our algorithm for synthesizing triggering terms that enable unsatisfiability proofs. We assume for simplicity that all quantified variables are globally unique and that the input formula `I` does not contain nested quantifiers. The auxiliary procedures `clustersRewritings` and `candidateTerm` are presented in Alg. 2 and Alg. 3, respectively.

Arguments : `I` — input formula, also treated as set of conjuncts
σ — similarity threshold for clustering
δ — maximum depth for clustering
μ — maximum number of different models
Result: The synthesized triggering term or None, if no term was found

1 **Procedure** synthesizeTriggeringTerm
2 **foreach** depth $\in \{0, \ldots, \delta\}$ **do**
3 **foreach** $F \in$ `I` $\wedge\, F \Leftrightarrow \forall \overline{x} :: F'$ **do**
4 **foreach** $(C, R) \in$ clustersRewritings(`I`, F, σ, depth) **do**
5 Inst $\longleftarrow \{\}$
6 **foreach** $f \in C \wedge f \Leftrightarrow (\forall \overline{x} :: D_0 \vee \ldots \vee D_n$ or $D_0 \vee \ldots \vee D_n)$ **do**
7 Inst$[f] \longleftarrow \{(\bigwedge_{0 \le j < k} \neg D_j) \wedge D_k \mid 0 \le k \le n\}$
8 Inst$[F] \longleftarrow \{\neg F'\}$
9 **foreach** $H \in \bigtimes \{\text{Inst}[f] \mid f \in \{F\} \cup C\}$ **do**
10 $G \longleftarrow \bigwedge H \wedge \bigwedge R$
11 **foreach** m $\in \{0, \ldots, \mu - 1\}$ **do**
12 resG, model $\longleftarrow$ checkSat(G)
13 **if** resG $\neq$ SAT **then**
14 **break** // No models if G is not SAT
15 $T \longleftarrow$ candidateTerm($\{F\} \cup C$, R, model) // Step 3
16 resI $\longleftarrow$ checkSat(`I` $\wedge\, T$) // Step 4
17 **if** resI $=$ UNSAT **then**
18 **return** minimized(T) // Success
19 $G \longleftarrow G \wedge \neg$model
20 **return** None

3.2 Algorithm

The pseudo-code of our algorithm is given in Alg. 1. It takes as input an SMT formula `I` (defined in Fig. 6), which we treat in a slight abuse of notation as both a formula and a set of conjuncts. Three other parameters allow us to customize the search strategy and are discussed later. The algorithm yields a triggering term that enables the unsat proof, or `None`, if no term was found. We assume here that `I` contains no nested quantifiers and present those in Appx. C.

The algorithm iterates over each *quantified* conjunct F of `I` (Alg. 1, line 3) and checks if F is individually unsatisfiable (for `depth = 0`). For complex proofs, this is usually not sufficient, as `I` is typically inconsistent due to *a combination* of conjuncts ($F_0 \wedge F_1$ in Fig. 5). In such cases, the algorithm proceeds as follows:

Algorithm 2: Auxiliary procedure for Alg. 1, which identifies clusters of formulas similar with F and their rewritings. `sim` is defined in text (step 1). `unify` is a first-order unification algorithm (not shown); it returns a set of rewritings with restrictive shapes, defined in text (step 2).

Arguments : `I` — input formula, also treated as set of conjuncts
F — quantified conjunct of `I`, i.e., $F \in \mathtt{I} \wedge F \Leftrightarrow \forall \overline{x} :: F'$
σ — similarity threshold for clustering
depth — current depth for clustering

Result: A set of pairs, consisting of clusters and their corresponding rewritings

1 **Procedure** clustersRewritings
2 **if** depth $= 0$ **then**
3 **return** $\{(\varnothing, \varnothing)\}$
4 similarFormulas $\longleftarrow \{f \mid f \in \mathtt{I} \setminus \{F\} \wedge \mathtt{sim}^{\mathsf{depth}}_{\mathtt{I}}(\mathrm{F}, \mathrm{f}, \sigma)\}$ // Step 1
6 rewritings $\longleftarrow \{\}$
7 **foreach** $f \in$ similarFormulas **do**
8 rws $\longleftarrow$ unify(F, f) // Step 2
9 **if** rws $= \varnothing \wedge \neg(f \Leftrightarrow D_0 \vee \ldots \vee D_n)$ **then**
10 similarFormulas $\longleftarrow$ similarFormulas $\setminus \{f\}$
11 rewritings$[f] \longleftarrow$ rws
12 **return** $\{(C, R) \mid C \subseteq$ similarFormulas $\wedge (\forall r \in R,\ \exists f \in C : r \in$ rewritings$[f])$
$\wedge (\forall x \in \mathtt{qvars}(C)\colon |\{r \mid r \in R \wedge x = \mathtt{lhs}(r)\}| \leq 1)\}$

Step 1: Clustering. It constructs clusters of formulas similar to F (Alg. 2, line 4), based on their *Jaccard similarity index*. Let F_i and F_j be two arbitrary formulas, and S_i and S_j their respective sets of uninterpreted function symbols (from their bodies and the patterns). The Jaccard similarity index is defined as:

$J(F_i, F_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$ (the number of common uninterpreted functions divided by the total number). For Fig. 5, $S_0 = \{\mathtt{f}\}$, $S_1 = \{\mathtt{f}, \mathtt{g}\}$, $J(F_0, F_1) = \frac{|\{\mathtt{f}\}|}{|\{\mathtt{f},\mathtt{g}\}|} = 0.5$.

Our algorithm explores the search space by iteratively expanding clusters to include transitively-similar formulas up to a maximum depth (parameter δ in Alg. 1). For two formulas $F_i, F_j \in \mathtt{I}$, we define the similarity function as:

$$\mathtt{sim}^{\delta}_{\mathtt{I}}(F_i, F_j, \sigma) = \begin{cases} J(F_i, F_j) \geq \sigma, & \delta = 1 \\ \exists F_k : \mathtt{sim}^{\delta-1}_{\mathtt{I}\setminus\{F_i\}}(F_i, F_k, \sigma) \wedge J(F_k, F_j) \geq \sigma, & \delta > 1 \end{cases}$$

where $\sigma \in [0, 1]$ is a similarity threshold used to parameterize our algorithm.

The initial cluster (`depth` $= 1$) includes all the conjuncts of `I` that are *directly* similar to F. Each subsequent iteration adds the conjuncts that are directly similar to an element of the cluster from the previous iteration, that is, *transitively* similar to F. This search strategy allows us to gradually strengthen the formulas G (used to synthesize candidate terms in step 3) without overly constraining them (an over-constrained formula is unsatisfiable, and has no models).

Step 2: Syntactic unification. The second step (Alg. 2, line 8) identifies *rewritings*, i.e., constraints under which two similar *quantified* formulas share terms.

(See Appx. D for quantifier-free formulas.) We obtain the rewritings by performing a *simplified* form of *syntactic term unification*, which reduces their number to a practical size. Our rewritings are *directed equalities*. For two formulas F_i and F_j and an uninterpreted function $\mathtt{f}$ they have one of the following shapes:

(1) $x_i = rhs_j$, where x_i is a quantified variable of F_i, rhs_j are terms from F_j defined below, F_i contains a term $\mathtt{f}(x_i)$ and F_j contains a term $\mathtt{f}(rhs_j)$,

(2) $x_j = rhs_i$, where x_j is a quantified variable of F_j, rhs_i are terms from F_i defined below, F_j contains a term $\mathtt{f}(x_j)$ and F_i contains a term $\mathtt{f}(rhs_i)$,

where rhs_k is a constant c_k, a quantified variable x_k, or a composite function $(\mathtt{f} \circ \mathtt{g}_0 \circ \ldots \circ \mathtt{g}_n)(\overline{c_k}, \overline{x_k})$ occurring in the formula F_k and $\mathtt{g}_0, \ldots, \mathtt{g}_n$ are arbitrary (interpreted or uninterpreted) functions. That is, we determine the *most general unifier* [6] only for those terms that have uninterpreted functions as the outermost functions and quantified variables as arguments. The unification algorithm is standard (except from the restrictive shapes), so it is not shown explicitly.

Since a term may appear more than once in F, or F unifies with multiple similar formulas through the same quantified variable, we can obtain *alternative rewritings* for a quantified variable. In such cases, we either duplicate or split the cluster, such that in each cluster-rewriting pair, each quantified variable is rewritten at most once (see Alg. 2, line 12). In Fig. 7, both F_1 and F_2 are similar to F_0 (all three formulas share the uninterpreted symbol $\mathtt{f}$). Since the unification produces alternative rewritings for x_0 ($x_0 = x_1$ and $x_0 = x_2$), the procedure `clustersRewritings` returns the pairs $\{(\{F_1\}, \{x_0 = x_1\}), (\{F_2\}, \{x_0 = x_2\})\}$.

$$F_0 \colon \forall x_0 \colon \text{Int} :: \{\mathtt{f}(x_0)\}\ \mathtt{f}(x_0) = 6$$
$$F_1 \colon \forall x_1 \colon \text{Int} :: \{\mathtt{f}(x_1)\}\ \mathtt{f}(x_1) = 7$$
$$F_2 \colon \forall x_2 \colon \text{Int} :: \{\mathtt{f}(x_2)\}\ \mathtt{f}(x_2) = 8$$

Fig. 7. Formulas that set contradictory constraints on the function $\mathtt{f}$. Synthesizing the triggering term $\mathtt{f}(0)$ requires clusters of similar formulas with alternative rewritings.

Step 3: Identifying candidate terms. From the clusters and the rewritings (identified before), we then derive *quantifier-free* formulas G (Alg. 1, line 10), and, if they are satisfiable, construct the candidate triggering terms from their models (Alg. 1, line 15). Each formula G consists of: (1) $\neg F_{\approx}$ (defined in (*), which is equivalent to $\neg F'$, since F has the shape $\forall \overline{x} :: F'$ from Alg. 1, line 3), (2) the *instantiations* (see (**)) of all the similar formulas from the cluster, and (3) the corresponding rewritings R. (Since we assume that all the quantified variables are globally unique, we do not perform variable renaming for the instantiations).

If a similar formula has multiple disjuncts D_k, the solver uses short-circuiting semantics when generating the model for G. That is, if it can find a model that satisfies the first disjunct, it does not consider the remaining ones. To obtain more diverse models, we synthesize formulas that *cover* each disjunct, i.e., make sure that it evaluates to true at least once. We thus compute *multiple instantiations* of

Algorithm 3: Auxiliary procedure for Alg. 1, which constructs a triggering term from the given cluster, rewritings, and SMT model. `dummy` is a fresh function symbol, which conveys no information about the truth value of the candidate term; thus conjoining it to the input preserves (un)satisfiability.

Arguments : C — set of formulas in the cluster
R — set of rewritings for the cluster
`model` — SMT model, mapping variables to values
Result: A triggering term with no semantic information

```
1 Procedure candidateTerm
2     P_0, ..., P_k ⟵ patterns(C)
3     while R ≠ ∅ do
4         choose and remove r ⟵ (x = rhs) from R
5         P_0, ..., P_k ⟵ (P_0, ..., P_k)[ rhs/x ]
6         R ⟵ R [ rhs/x ]
7     foreach x ∈ qvars(C) do
8         P_0, ..., P_k ⟵ (P_0, ..., P_k)[ model(x)/x ]
9     return "dummy" + "(" + P_0, ..., P_k + ")"
```

each similar formula, of the form: $(\bigwedge_{0 \le j < k} \neg D_j) \wedge D_k, \forall k\colon 0 \le k \le n$ (see Alg. 1, line 7). To consider all the combinations of disjuncts, we derive the formula G from the Cartesian product of the instantiations (Alg. 1, line 9). (To present the pseudo-code in a concise way, we store $\neg F'$ in the instantiations map as well (Alg. 1, line 8), even if it does *not* represent the instantiation of F.)

In Fig. 8, F_1 is similar to F_0 and $R = \{x_0 = x_1\}$. F_1 has two disjuncts and thus two possible instantiations: $\mathsf{Inst}[F_1] = \{x_1 \ge 1, (x_1 < 1) \wedge (\mathtt{f}(x_1) = 6)\}$. The formula $G = (x_0 > -1) \wedge (\mathtt{f}(x_0) \le 7) \wedge (x_1 \ge 1) \wedge (x_0 = x_1)$ for the first instantiation is satisfiable, but none of the values the solver can assign to x_0 (which are all greater or equal to 1) are sufficient for the unsatisfiability proof to succeed. The second instantiation adds additional constraints: instead of $x_1 \ge 1$, it requires $(x_1 < 1) \wedge (\mathtt{f}(x_1) = 6)$. The resulting G formula has a unique solution for x_0, namely 0, and the triggering term $\mathtt{f}(0)$ is sufficient to prove unsat.

$$F_0\colon \forall x_0\colon \mathrm{Int} :: \{\mathtt{f}(x_0)\}\ \neg(x_0 > -1) \vee \mathtt{f}(x_0) > 7$$
$$F_1\colon \forall x_1\colon \mathrm{Int} :: \{\mathtt{f}(x_1)\}\ \neg(x_1 < 1) \vee \mathtt{f}(x_1) = 6$$

Fig. 8. Formulas that set contradictory constraints on the function `f`. Synthesizing the triggering term $\mathtt{f}(0)$ requires instantiations that cover all the disjuncts.

The procedure candidateTerm in Alg. 3 synthesizes a candidate triggering term T from the models of G and the rewritings R. We first collect all the patterns of the formulas from the cluster C (Alg. 3, line 2), i.e., of F and of its similar conjuncts (see Alg. 1, line 15). Then, we *apply* the rewritings, in an arbitrary order (Alg. 3, lines 3–6). That is, we substitute the quantified variable

x from the left hand side of the rewriting with the right hand side term rhs and propagate this substitution to the remaining rewritings. This step allows us to include in the synthesized triggering terms additional information, which cannot be provided by the solver. Then (Alg. 3, lines 7–8) we substitute the remaining variables with their *constant* values from the model (i.e., constants for built-in types, and fresh, unconstrained variables for uninterpreted types). The resulting triggering term is wrapped in an application to a fresh, uninterpreted function `dummy` to ensure that conjoining it to `I` does not change `I`'s satisfiability.

Step 4: Validation. We validate the candidate triggering term T by checking if $\texttt{I} \wedge T$ is unsatisfiable, i.e., if these particular interpretations for the uninterpreted functions generalize to all interpretations (Alg. 1, line 16). If this is the case then we return the *minimized* triggering term (Alg. 1, line 18). The `dummy` function has multiple arguments, each of them corresponding to one pattern from the cluster (Alg. 3, line 9). This is an over-approximation of the required triggering terms (once instantiated, the formulas may trigger each other), so `minimized` removes redundant (sub-)terms. If T does not validate, we re-iterate its construction up to a bound μ and strengthen the formula G to obtain a different model (Alg. 3, lines 19 and 11). Appx. B discusses heuristics for obtaining *diverse models*.

4 Evaluation

Evaluating our work requires benchmarks with known triggering issues. As there is no publicly available suite, in Sec. 4.1 we used manually-collected benchmarks from four verifiers [20,37,17,26]. Our algorithm succeeded for 65,62%. To evaluate its applicability to other verifiers, in Sec. 4.2 we used SMT-COMP [35] inputs. As they were not designed to expose triggering issues, we developed a pre-filtering step (Appx. G) to automatically identify the subset that falls into this category. The results show that our algorithm is suited also for benchmarks from [8,33,11]. Sec. 4.3 illustrates that our triggering terms are simpler than unsat proofs, which are sometimes produced by quantifier instantation and refutation techniques.

Setup. We used Z3 (4.8.10) [25] to infer the patterns, generate the models and validate the candidate terms. However, our tool can be used with any solver that supports E-matching and exposes the inferred patterns. We used Z3's `NNF` tactic to transform the inputs into NNF and locality-sensitive hashing to compute the clusters. We fixed Z3's random seeds to arbitrary values (`sat.random_seed` to 488, `smt.random_seed` to 599, and `nlsat.seed` to 611). We set the (soft) timeout to 600s and the memory limit to 6 GB per run and used a 1s timeout for obtaining a model and for validating a candidate term. The experiments were conducted on a Linux server with 252 GB of RAM and 32 Intel Xeon CPUs at 3.3 GHz.

4.1 Effectiveness on verification benchmarks with triggering issues

First, we used manually-collected benchmarks with known triggering issues from Dafny [20], F* [37], Gobra [17], and Viper [26]. We reconstructed 4, respectively

Table 1. Results on verification benchmarks with known triggering issues. The columns show: the source of the benchmarks, the number of files (#), their number of conjuncts ($\#F$) and of quantifiers ($\#\forall$), the number of files for which five configurations (C0–C4) synthesized suited triggering terms, our results across all configurations, the number of unsat proofs generated by Z3 (with MBQI [16]), CVC4 (with enumerative instantiation [29]), and Vampire [19] (in CASC mode [36], using Z3 for ground theory reasoning).

Source	#	$\#F$	$\#\forall$	C0 default	C1 σ=0.1	C2 β=1	C3 type	C4 σ=0.1 $\wedge$ sub	Our work	Z3 MBQI	CVC4 enum inst	Vampire CASC $\wedge$ Z3
Dafny	4	6 - 16	5 - 16	1	1	1	1	0	**1**	1	0	2
F*	2	18 - 2388	15 - 2543	1	1	1	1	2	**2**	1	0	2
Gobra	11	64 - 78	50 - 63	5	10	1	7	10	**11**	6	0	11
Viper	15	84 - 143	68 - 203	7	5	3	5	5	**7**	11	0	15
Total out of	**32**			**21 (65,62%)**						**19 (59,37%)**	**0 (0%)**	**30 (93,75%)**

σ = similarity threshold **β** = batch size **type** = type-based constraints **sub** = sub-terms **C0**: $\sigma = 0.3$; $\beta = 64$; $\neg$**type**; $\neg$**sub**

2 inconsistent axiomatizations from Dafny and F*, based on the changes from the repositories and the messages from the issue trackers; we obtained 11 inconsistent axiomatizations of arrays and option types from Gobra's developers and collected 15 incompleteness issues from Viper's test suite [3], with at least one assertion needed only for triggering. These contain algorithms for arrays, binomial heaps, binary search trees, and regression tests. The file sizes (minimum-maximum number of formulas or quantifiers) are shown in Tab. 1, columns 3–4.

Configurations. We ran our tool with five configurations, to also analyze the impact of its parameters (see Alg. 1 and Appx. C). The default configuration C0 has: $\sigma = 0.3$ (similarity threshold), $\beta = 64$ (batch size, i.e., the number of candidate terms validated together), ¬type (no type-based constraints), ¬sub (no unification for sub-terms). The other configurations differ from C0 in the parameters shown in Tab. 1. All configurations use $\delta = 2$ (maximum transitivity depth), $\mu = 4$ (maximum number of different models), and 600s timeout per file.

Results. Columns 5–9 in Tab. 1 show the number of files solved by each configuration, column 10 summarizes the files solved by at least one. Overall, we synthesized suited triggering terms for 65,62%, including all F* and Gobra benchmarks. An F* unsoundness exposed by all configurations in ≈60s is presented in Fig. 9. It required two developers to be manually diagnosed based on a bug report [2]. A simplified Gobra axiomatization for option types, solved by C4 in ≈13s, is shown in Fig. 12. Gobra's team spent one week in identifying some of the issues. As our triggering terms for F* and Gobra were similar to the manually-written ones, they could have reduced the human effort in localizing and fixing the errors.

Our algorithm synthesized missing triggering terms for 7 Viper files, including the array maximum example [1], for which E-matching could not prove that the maximal element in a strictly increasing array of size 3 is its last element. Our triggering term `loc(a,2)` (`loc` maps arrays and integers to heap locations) can be added by a *user* of the verifier to their postcondition. A *developer* can fix the root cause of the incompleteness by including a generalization of the triggering term to arbitrary array sizes: `len(a)!=0 ==> x == loc(a, len(a)-1).val`.

Table 2. Results on SMT-COMP inputs. The columns have the structure from Tab. 1.

Source	#	#*F*	#∀	C0 default	C1 σ-0.1	C2 β-1	C3 type	C4 σ-0.1 ∧ sub	Our work	Z3 MBQI	CVC4 enum inst	Vampire CASC ∧ Z3
Spec#	33	28 - 2363	25 - 645	16	16	14	16	15	**16**	16	0	29
VCC/Havoc	14	129 - 1126	100 - 1027	11	9	5	11	9	**11**	12	0	14
Simplify	1	256	129	0	0	0	0	0	**0**	1	0	0
BWI	13	189 - 384	198 - 456	1	1	2	1	1	**2**	12	0	12
Total out of	**61**					**29 (47,54%)**				**41 (67,21%)**	**0 (0%)**	**55 (90,16%)**

σ = similarity threshold **β** = batch size **type** = type-based constraints **sub** = sub-terms **C0**: **σ = 0.3**; **β = 64**; **¬type**; **¬sub**

Either solution results in E-matching refuting the proof obligation in under 0.1s. Our tool also exposed another case where Boogie [7] (which is used by Viper) is only sound modulo patterns (similar to the one from Fig. 3 [23]).

4.2 Effectiveness on SMT-COMP benchmarks

Next, we considered 61 SMT-COMP [35] benchmarks from Spec# [8], VCC [33], Havoc [11], Simplify [13], and the Bit-Width-Independent (BWI) encoding [27].

Results. The results are shown in Tab. 2. Our algorithm enabled E-matching to refute 47.54% of the files, most of them from Spec# and VCC/Havoc. We manually inspected some BWI benchmarks (for which the algorithm had worse results) and observed that the validation step times out even with a much higher timeout. This shows that some candidate terms trigger matching loops and explains why C2 (which validates them individually) solved one more file. Extending our algorithm to avoid matching loops, by construction, is left as future work.

4.3 Comparison with unsatisfiability proofs

As an alternative to our approach, tool developers could try to manually identify triggering issues from refutation proofs, but these are usually very complex. Columns 11–13 in Tab. 1 and Tab. 2 show the number of proofs produced by Z3 with MBQI [16], CVC4 [9] with enumerative instantiation [29], and Vampire [19] using Z3 for ground theory reasoning [28] and the CASC [36] portfolio mode with competition presets. CVC4 failed for all examples (it cannot construct proofs for quantified logics), Vampire refuted most of them. Our algorithm outperformed MBQI for F* and Gobra and had similar results for Dafny, Spec# and VCC/Havoc. All C0–C4 solved two VCC/Havoc files not solved by MBQI (see Appx. D). Moreover, MBQI's proof for Viper's array maximum example has 2135 lines and over 700 reasoning steps; Vampire's proof has 348 lines and 101 inference steps. Other proofs have similar complexity. Our triggering term, `loc(a,2)`, is much simpler and, thus, much easier to understand.

Even though Vampire successfully refuted most of the benchmarks, it is not a replacement for our technique. First, most program verifiers employ SMT solvers with E-matching; it is thus important to help the developers to correctly use the technology of their choice. Second, Vampire would make unsound those verifiers that rely on E-matching for soundness, as we have illustrated for Boogie.

5 Related Work

To our knowledge, no other approach automatically produces the information needed by developers to remedy the effects of overly restrictive patterns. Quantifier instantiation and refutation techniques (discussed next) can produce unsatisfiability proofs, but these are much more complex than our triggering terms.

Quantifier instantiation techniques. Model-based quantifier instantiation [16] (MBQI) was designed for sat formulas. It checks if the models obtained for the quantifier-free part of the input satisfy the quantifiers, whereas we check if the synthesized triggering terms obtained for some interpretation of the uninterpreted functions generalize to all interpretations. In some cases, MBQI can also generate unsatisfiability proofs, but they require expert knowledge to be understood; our triggering terms are much simpler. *Counterexample-guided quantifier instantiation* [30] is a technique for sat formulas, which synthesizes computable functions from logical specifications. It is applicable to functions whose specifications have explicit syntactic restrictions on the space of possible solutions, which is usually not the case for axiomatizations. Thus the technique cannot directly solve the complementary problem of proving soundness of the axiomatization.

E-matching-based approaches. Rümmer [32] proposed a *calculus* for first-order logic modulo linear integer arithmetic that integrates constraint-based free variable reasoning with E-matching. Our algorithm does not require reasoning steps, so it is applicable to formulas from all the logics supported by the SMT solver. *Enumerative instantiation* [29] is an approach that exhaustively enumerates ground terms from a set of ordered, quantifier-free terms from the input. It can be used to refute formulas with quantifiers, but not to construct proofs (see Sec. 4.3). Our algorithm derives quantifier-free formulas and synthesizes the triggering terms from their models, even if the input does not have a quantifier-free part. It uses also syntactic information to construct complex triggering terms.

Theorem provers. First-order theorem provers (e.g., Vampire [19]) also generate refutation proofs. More recent works combine a superposition calculus with theory reasoning [39,28], integrating SAT/SMT solvers with theorem provers. We also use unification, but to synthesize triggering terms required by E-matching. However, our triggering terms are much simpler than Vampire's proofs and can be used to improve the triggering strategies for all future runs of the verifier.

6 Conclusions

We have presented the first automated technique that enables the developers of verifiers remedy the effects of overly restrictive patterns. Since discharging proof obligations and identifying inconsistencies in axiomatizations require an SMT solver to prove the unsatisfiability of a formula via E-matching, we developed a novel algorithm for synthesizing triggering terms that allow the solver to complete the proof. Our approach is effective for a diverse set of verifiers, and can significantly reduce the human effort in localizing and fixing triggering issues.

References

1. Array maximum, by elimination (2021), http://viper.ethz.ch/examples/max-array-elimination.html
2. F* issue 1848 (2021), https://github.com/FStarLang/FStar/issues/1848
3. Viper test suite (2021), https://github.com/viperproject/silver/tree/master/src/test/resources
4. Amighi, A., Blom, S., Huisman, M.: Vercors: A layered approach to practical verification of concurrent software. In: PDP. pp. 495–503. IEEE Computer Society (2016), https://ieeexplore.ieee.org/abstract/document/7445381
5. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). vol. 3, pp. 147:1–147:30. ACM (2019). https://doi.org/10.1145/3360573
6. Baader, F., Snyder, W.: Unification theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning. pp. 445–532. Elsevier and MIT Press (2001)
7. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects (FMCO). Lecture Notes in Computer Science, vol. 5, pp. 364–387. Springer (2005)
8. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The spec# experience. Communications of the ACM **54**(6), 81–91 (June 2011)
9. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
10. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
11. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 19–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
12. Darvas, A., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Dwyer, M.B., Lopes, A. (eds.) Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 4422, pp. 336–351. Springer-Verlag (2007)
13. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (May 2005). https://doi.org/10.1145/1066100.1066102, http://doi.acm.org/10.1145/1066100.1066102
14. Eilers, M., Müller, P.: Nagini: A static verifier for python. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification (CAV). LNCS, vol. 10982, pp. 596–603. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-96145-3_33
15. Gario, M., Micheli, A.: Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In: SMT Workshop 2015 (2015)

16. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. pp. 306–320. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
17. Gobra Team: Gobra: An automated, modular verifier for Go programs (2020), https://github.com/viperproject/gobra
18. Heule, S., Kassios, I.T., Müller, P., Summers, A.J.: Verification condition generation for permission logics with abstract predicates and abstraction functions. In: Castagna, G. (ed.) European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 7920, pp. 451–476. Springer (2013)
19. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 1–35. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
20. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
21. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order smt solvers. In: Proceedings of the 2009 ACM Symposium on Applied Computing. pp. 615–622. SAC'09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1529282.1529411, https://doi.org/10.1145/1529282.1529411
22. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 4960, pp. 307–321. Springer-Verlag (2008)
23. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 312–327. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
24. Moskal, M.: Programming with triggers. In: SMT. ACM International Conference Proceeding Series, vol. 375, pp. 20–29. ACM (2009)
25. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
26. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 9583, pp. 41–62. Springer-Verlag (2016)
27. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards bit-width-independent proofs in smt solvers. In: Fontaine, P. (ed.) Automated Deduction – CADE 27. pp. 366–384. Springer International Publishing, Cham (2019)
28. Reger, G., Bjorner, N., Suda, M., Voronkov, A.: Avatar modulo theories. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) GCAI 2016. 2nd Global Conference on Artificial Intelligence. EPiC Series in Computing, vol. 41, pp. 39–52. EasyChair (2016). https://doi.org/10.29007/k6tp, https://easychair.org/publications/paper/7
29. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 112–131. Springer International Publishing, Cham (2018)
30. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in smt. In: Kroening, D., Păsăreanu,

C.S. (eds.) Computer Aided Verification. pp. 198–216. Springer International Publishing, Cham (2015)
31. Rudich, A., Ádám Darvas, Müller, P.: Checking well-formedness of pure-method specifications. In: Cuellar, J., Maibaum, T. (eds.) Formal Methods (FM). Lecture Notes in Computer Science, vol. 5014, pp. 68–83. Springer-Verlag (2008)
32. Rümmer, P.: E-matching with free variables. In: Bjørner, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 359–374. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
33. Schulte, W.: Vcc: Contract-based modular verification of concurrent c. In: 31st International Conference on Software Engineering, ICSE 2009. IEEE Computer Society (January 2008), https://www.microsoft.com/en-us/research/publication/vcc-contract-based-modular-verification-of-concurrent-c/
34. SMT-COMP: The 14th international satisfiability modulo theories competition (including pending benchmarks) (2019), https://smt-comp.github.io/2019/,https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks-tmp/benchmarks-pending
35. SMT-COMP: The 15th international satisfiability modulo theories competition (2020), https://smt-comp.github.io/2020/
36. Sutcliffe, G.: The CADE ATP System Competition - CASC. AI Magazine **37**(2), 99–101 (2016)
37. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in f*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 256–270. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837655, https://doi.org/10.1145/2837614.2837655
38. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. In: Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation. pp. 387–398. PLDI '13 (2013), https://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/
39. Voronkov, A.: Avatar: The architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 696–710. Springer International Publishing, Cham (2014)

Appendix A Background: E-matching

In this section, we briefly discuss the E-matching-related terminology and explain how this quantifier-instantiation algorithm works on an example.

Patterns vs triggering terms. Patterns are syntactic hints attached to quantifiers which instruct the SMT solver when to perform an instantiation. In Fig. 2, the quantified formula F_3 will be instantiated only when a *triggering term* that matches the *pattern* $\{\texttt{Length}(s_3)\}$ is encountered during the SMT run (i.e., the triggering term is present in the quantifier-free part of the input formula or is obtained by the solver from the body of a previously-instantiated quantifier).

E-matching. We now illustrate how E-matching works on the example from Fig. 2; in particular, we show how our synthesized triggering term $\texttt{Length}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))$ helps the solver to prove unsat when added to the axiomatization (v is a fresh variable of type U). Due to space constraints, we omit unnecessary instantiations. The sub-terms $\texttt{Empty}(\texttt{typ}(v))$ and $\texttt{Length}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))$ trigger the instantation of F_1 and F_4, respectively. The solver obtains the body of the quantifiers for these particular values:

$$
\begin{aligned}
B_1 &: \texttt{typ}(\texttt{Empty}(\texttt{typ}(v))) = \texttt{Type}(\texttt{typ}(v)) \\
B_4 &: \neg(\texttt{typ}(\texttt{Empty}(\texttt{typ}(v))) = \texttt{Type}(\texttt{typ}(v))) \vee \\
&\quad (\texttt{Length}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)) = -1)
\end{aligned}
$$

Since the first disjunct of B_4 evaluates to `false` (from B_1), the solver learns that the second disjunct must hold (i.e., the length must be -1); we abbreviate it as L = -1. Further, the sub-terms $\texttt{Build}(\texttt{Empty}(\texttt{typ}(v))$ and $\texttt{Length}(\texttt{SeqBuild}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))$ of the synthesized triggering term lead to the instantiation of F_2 and F_3, respectively:

$$
\begin{aligned}
B_2 &: \texttt{type}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)) = \texttt{Type}(\texttt{typ}(v)) \\
B_3 &: \neg(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)) = \\
&\quad \texttt{Type}(\texttt{ElemType}(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))))) \vee \\
&\quad (0 \leq \texttt{Length}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))))
\end{aligned}
$$

$\texttt{Type}(\texttt{ElemType}(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))))$ from B_3 triggers F_0:

$$
\begin{aligned}
B_0 &: \texttt{ElemType}(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))) = \\
&\quad \texttt{ElemType}(\texttt{Type}(\texttt{ElemType}(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))))))
\end{aligned}
$$

By equalizing the arguments of the outer-most `ElemType` in B_0, the solver learns that the first disjunct of B_3 is `false`. The second disjunct must thus hold (i.e., the length should be positive); we abbreviate it as $0 \leq L$. Since $(L = -1) \wedge (0 \leq L) = \texttt{false}$, the unsatisfiability proof succeeds.

Appendix B Diverse models

In this section, we explain the importance of the parameter μ from Alg. 1 (the maximum number of models) and discuss heuristics for obtaining diverse models.

$$F\colon \forall x\colon \text{Int}, y\colon \text{Int} :: \{\texttt{\_div}(x,y)\}\ \texttt{\_div}(x,y) = x/y$$

Fig. 9. Inconsistent axiom from F* [37]. `_div`: Int × Int → Int is an uninterpreted function. Synthesizing the triggering term `dummy`(`_div`(1, 2)) requires diverse models.

Let us consider the formula from Fig. 9, which was part of an axiomatization with 2,495 axioms. F axiomatizes the uninterpreted function `_div`: Int × Int → Int and is inconsistent, because there exist two integers whose real division ("/") is not an integer. The model produced by the solver for the formula $G = \neg F'$ is $x = -1, y = 0$. $-1/0$ is defined ("/" is a total function [10]), but its result is not specified. Thus the solver cannot validate this model (i.e., it returns *unknown*).

In such cases or when the candidate term does not generalize to all interpretations of the uninterpreted functions, we re-iterate its construction, up to the bound μ (Alg. 1, line 11). For this, we strengthen the previously-derived formula G to force the solver find a different model. In Fig. 9, if we simply exclude previous models, we can obtain a sequence of models with different values for the numerator, but with the same value (0) for the denominator. There are infinitely many such models, and all of them fail to validate for the same reason.

There are various heuristics one can employ to guide the solver's search for a new model and our algorithm can be parameterized with different ones. In our experiments, we interpret the conjunct $\neg\texttt{model}$ from Alg. 1, line 19 as $\bigwedge_{x\in\overline{x}} x \neq \texttt{model}(x) \wedge \bigwedge_{x_i,x_j\in\overline{x},\ i\neq j,\ \texttt{model}(x_i)=\texttt{model}(x_j)} x_i \neq x_j$). The first component requires all the variables to have different values than before. This requirement may be too strong for some variables, but as we use only *soft* constraints, the solver may ignore some constraints if it cannot generate a satisfying assignment.

The second part requires models from different *equivalence classes*, where an equivalence class includes all the variables that are equal in the model. For example, if the model is $x_0 = x, x_1 = x$, where x is a value of the corresponding type, then x_0 and x_1 belong to the same equivalence class. Considering equivalence classes is particularly important for variables of uninterpreted types; the solver cannot provide actual values for them, thus it assigns fresh, unconstrained variables. However, different fresh variables do not lead to diverse models.

Appendix C Extensions

Next, we describe various extensions of our algorithm that enable complex proofs.

Nested quantifiers. Nested existential quantifiers in positive positions and nested universal quantifiers in negative positions are replaced in NNF by new, uninterpreted Skolem functions. The unification (Sec. 3.2, step 2) is also applicable to them. Skolem functions with arguments (i.e., the quantified variables from the outer scope) are unified as regular uninterpreted functions. They can also appear as *rhs* in a rewriting, but not as the left-hand side (we do not perform higher-order unification). In such cases, our algorithm is imprecise. The unification of $\texttt{f}(x_0, \texttt{skolem}())$ and $\texttt{f}(x_1, 1)$, for example, produces only the rewriting $x_0 = x_1$.

After pre-processing, the conjunct F and the similar formulas may still contain nested universal quantifiers. F is always negated in G, thus it becomes, after Skolemization, quantifier-free. To ensure that G is also quantifier-free (and the solver can generate a model for it), our extended algorithm *recursively instantiates* similar formulas with nested quantifiers when computing the instantiations.

$F_0 \colon \forall l_0 \colon \mathrm{L} :: \{\texttt{isEmpty}(l_0)\}\ \neg(l_0 = \texttt{EmptyList}) \vee \texttt{isEmpty}(l_0)$

$F_1 \colon \forall l_1 \colon \mathrm{L} :: \{\texttt{isEmpty}(l_1)\}\ \texttt{isEmpty}(l_1) \vee \texttt{has}(l_1, \texttt{f}_1(l_1))$

$F_2 \colon \forall l_2 \colon \mathrm{L} :: \{\texttt{isEmpty}(l_2)\}\ \neg\texttt{isEmpty}(l_2) \vee \forall el_2 \colon \mathrm{Int} :: \{\texttt{has}(l_2, el_2)\}\ \neg\texttt{has}(l_2, el_2)$

$F_3 \colon \forall l_3 \colon \mathrm{L}, el_3 \colon \mathrm{Int} :: \{\texttt{has}(l_3, el_3)\}\ \texttt{has}(l_3, el_3) \vee (\texttt{indexOf}(l_3, el_3) = -1)$

$F_4 \colon \forall l_4 \colon \mathrm{L}, el_4 \colon \mathrm{Int} :: \{\texttt{indexOf}(l_4, el_4)\}\ \texttt{indexOf}(l_4, el_4) \geq 0$

Fig. 10. Formulas that set contradictory constraints on `indexOf`. L is an uninterpreted type, `EmptyList` is a user-defined constant of type L. $\texttt{f}_1$ is a Skolem function, which replaces a nested existential quantifier. F_2 contains nested universal quantifiers.

In Fig. 10, F_0–F_4 axiomatize operations over lists of integers. The axioms F_3 and F_4 set contradictory constraints on `indexOf` when the element is not contained in the list. According to Alg. 2, one of the clusters generated for F_3 is $C = \{F_2, F_0\}$, with the rewritings $R = \{l_3 = l_2, el_3 = el_2, l_2 = l_0\}$. The algorithm then computes the instantiations for F_0 and F_2; as F_2 contains nested quantifiers, we remove both of them and obtain: $\mathsf{Inst}[F_2] = \{\neg\texttt{isEmpty}(l_2), \texttt{isEmpty}(l_2) \wedge \neg\texttt{has}(l_2, el_2)\}$, $\mathsf{Inst}[F_0] = \{\neg(l_0 = \texttt{EmptyList}), l_0 = \texttt{EmptyList} \wedge \texttt{isEmpty}(l_0)\}$. The model of the corresponding G formula and R allow us to synthesize the required triggering term $T = \texttt{dummy}(\texttt{isEmpty}(\texttt{EmptyList}), \texttt{has}(\texttt{EmptyList}, 0))$.

Combining multiple candidate terms. In Alg. 1, each candidate term is validated separately. To enable proofs that require multiple instantiations of the *same* formula, we developed an extension that validates multiple triggering terms at the same time. In such cases, the algorithm returns a *set of terms* that are necessary and sufficient to prove unsat. Fig. 11 presents a simple example from SMT-COMP 2019 pending benchmarks [34]. The input $\texttt{I} = F_0 \wedge F_1$ is unsatisfiable, as there does not exist an interpretation for the function `U` that satisfies all the constraints: F_1 requires $\texttt{U}(\texttt{s})$ to be true; if F_0 is instantiated for $x_0 = \texttt{s}$, the solver learns that $\texttt{U}(\texttt{il})$ must be true as well; however, if $x_0 = \texttt{il}$, then $\texttt{U}(\texttt{il})$

must be false, which is a contradiction. Exposing the inconsistency thus requires two instantiations of F_0, triggered by $\texttt{f}(\texttt{s})$ and $\texttt{f}(\texttt{il})$, respectively. We generate both triggering terms, but in separate iterations (independently, both fail to validate). However, by validating them simultaneously (i.e., conjoin both of them to $\texttt{I}$), our algorithm identifies the required triggering term $T = \texttt{dummy}(\texttt{f}(\texttt{s}), \texttt{f}(\texttt{il}))$.

$$F_0 \colon \forall x_0 \colon \mathrm{S} :: \{\texttt{f}(x_0)\}\ \neg\texttt{U}(x_0) \vee (\texttt{U}(\texttt{f}(x_0)) \wedge \texttt{f}(x_0) = \texttt{il} \wedge x_0 \neq \texttt{il})$$
$$F_1 \colon \texttt{U}(\texttt{s})$$

Fig. 11. Benchmark from SMT-COMP 2019 [34]. The formulas set contradictory constraints on the function `U`. S is an uninterpreted type, `s` and `il` are user-defined constants of type S. Synthesizing the triggering term $\texttt{dummy}(\texttt{f}(\texttt{s}), \texttt{f}(\texttt{il}))$ requires multiple candidate terms. We use conjunctions here for simplicity, but our pre-processing applies distributivity of disjunction over conjunction and splits F_0 into three different formulas with unique names for the quantified variables.

Unification across multiple instantiations. The clusters constructed by our algorithm are sets (see Alg. 2, line 12), so they contain a formula at most once, even if it is similar to multiple other formulas from the cluster. We thus consider the rewritings for multiple instantiations of the same formula separately, in different iterations. To handle cases that require multiple (but boundedly many) instantiations, we extend the algorithm with a parameter Φ, which bounds the maximum frequency of a *quantified* conjunct within the formulas G. That is, it allows a similar quantified formula, as well as F itself, to be added to a cluster more than once (after performing variable renaming, to ensure that the names of the quantified variables are still globally unique). This results in an equisatisfiable formula for which our algorithm determines multiple triggering terms. Inputs whose unsatisfiability proofs require an unbounded number of instantiations typically contain a matching loop, thus we do not consider them here.

$$F_0 \colon \forall e_0 \colon \mathrm{U} :: \{\texttt{some}(e_0)\}\ \neg(\texttt{some}(e_0) = \texttt{none})$$
$$F_1 \colon \forall op_1 \colon \mathrm{U}, e_1 \colon \mathrm{U} :: \{\texttt{some}(e_1), \texttt{get}(op_1)\}\ \neg(\texttt{get}(op_1) = e_1) \vee (op_1 = \texttt{some}(e_1))$$
$$F_2 \colon \forall op_2 \colon \mathrm{U}, e_2 \colon \mathrm{U} :: \{\texttt{some}(e_2), \texttt{get}(op_2)\}\ \neg(op_2 = \texttt{some}(e_2)) \vee (\texttt{get}(op_2) = e_2)$$

Fig. 12. Fragment of Gobra's option types axiomatization. U is an uninterpreted type, `none` is a user-defined constant of type U. $F_1 - F_2$ have *multi-patterns* (Appx. D). Synthesizing the triggering term $\texttt{dummy}(\texttt{some}(\texttt{get}(\texttt{none})))$ requires type-based constraints.

Type-based constraints. The rewritings of the form $x_i = x_j$ can be too imprecise (especially for quantified variables of uninterpreted types), as they do not constrain the *rhs*. In Fig. 12, the solver cannot provide concrete values of type U

for e_1 and op_1, it can only assign fresh, unconstrained variables (e.g., e and op). However, the triggering terms $\texttt{some}(e)$ and $\texttt{get}(op)$ are not sufficient to prove unsat; one would additionally need the rewriting $e_1 = \texttt{get}(op_1)$, which cannot be identified by our unification from Sec. 3.2. To address such scenarios, we extend the unification to also consider as *rhs* a constant or an uninterpreted function from the body of the similar formulas, which has the same type as the quantified variable from the left-hand side. For Fig. 12, it will thus generate the rewritings $R = \{e_0 = \texttt{get}(op_2), e_1 = \texttt{get}(op_2), op_1 = \texttt{none}, op_2 = \texttt{none}\}$ (this is one of the alternatives). These type-based constraints allow us to synthesize the triggering term $T = \texttt{dummy}(\texttt{some}(\texttt{get}(\texttt{none})))$, which exposes the unsoundness from Gobra's option types axiomatization.

Unification for sub-terms. Fig. 13 shows an example which cannot be solved by any extension discussed so far, since it requires semantic reasoning: by applying `f` on both sides of the equality, one can learn from F_1 that $\texttt{f}(\texttt{g}(2020)) = \texttt{f}(\texttt{g}(2021))$. From F_0 though, $\texttt{f}(\texttt{g}(2020)) = 2020$ and $\texttt{f}(\texttt{g}(2021)) = 2021$, which implies that $2020 = 2021$, i.e., false. Our extended algorithm synthesizes the required triggering term $T = \texttt{dummy}(\texttt{f}(\texttt{g}(2020)), \texttt{f}(\texttt{g}(2021)))$ by applying the unification also to *sub-terms*. In Fig. 13, trying to unify $\texttt{f}(\texttt{g}(x_0))$ does not produce any rewritings, as F_1 does not contain $\texttt{f}(\texttt{g})$. We thus unify the subterm $\texttt{g}(x_0)$ with $\texttt{g}(2020)$ and $\texttt{g}(2021)$ and obtain the rewritings $R = \{x_0 = 2020, x_0 = 2021\}$. Together with the extension for combining multiple candidate terms described above, these rewritings provide sufficient information for the unsatisfiability proof to succeed.

$$F_0 \colon \forall x_0 \colon \text{Int} :: \{\texttt{f}(\texttt{g}(x_0))\}\ \texttt{f}(\texttt{g}(x_0)) = x_0$$
$$F_1 \colon \texttt{g}(2020) = \texttt{g}(2021)$$

Fig. 13. Formulas that set contradictory constraints on the function `f`. Synthesizing the triggering term $\texttt{dummy}(\texttt{f}(\texttt{g}(2020)), \texttt{f}(\texttt{g}(2021)))$ requires unification for sub-terms.

Alternative triggering terms. Our algorithm returns the *first* candidate term that successfully validates (Alg. 1, line 18). However, it might also be useful to synthesize *alternative* triggering terms for the same input, which may correspond to different completeness or soundness issues. Our tool provides this option.

Appendix D Additional Examples

In this section, we illustrate our algorithm on various examples (including those from Fig. 1 and Fig. 2), and discuss how it supports quantifier-free formulas, external patterns, multi-patterns and alternative patterns.

Quantifier-free formulas. Our algorithm from Alg. 1 iterates only over quantified conjuncts, but leverages the additional information provided by quantifier-free formulas and includes them in the clusters even if the unification cannot find a rewriting (Alg. 2, line 9). Since quantifier-free conjuncts can be seen as already instantiated formulas, we only have to cover all their disjuncts (Alg. 1, line 7).

$$F_0 : \forall x_0 : \text{Int} :: \{\texttt{len}(\texttt{nxt}(x_0))\}\ \texttt{len}(x_0) > 0$$
$$F_1 : \forall x_1 : \text{Int} :: \{\texttt{len}(\texttt{nxt}(x_1))\}\ (\texttt{nxt}(x_1) = x_1) \vee (\texttt{len}(x_1) = \texttt{len}(\texttt{nxt}(x_1)) + 1)$$
$$F_2 : \forall x_2 : \text{Int} :: \{\texttt{len}(\texttt{nxt}(x_2))\}\ \neg(\texttt{nxt}(x_2) = x_2) \vee (\texttt{len}(x_2) = 1)$$
$$F_3 : \texttt{len}(7) \leq 0$$

Fig. 14. Boogie example from Fig. 1 encoded in our input format. F_0–F_2 represent the axiom, while the quantifier-free formula F_3 is the negation of the assertion (verifiers discharge their proof obligations by showing that the negation is unsatisfiable).

Boogie example. Fig. 14 shows the Boogie example from Fig. 1 in our format. The quantifier-free formula F_3 (i.e., the verification condition) is similar with F_0 (they share the symbol `len`) and unifies through the rewritings $R = \{x_0 = 7\}$. We obtain the required triggering term $T = \texttt{dummy}(\texttt{len}(\texttt{nxt}(7)))$ from the model of the formula $G = \neg F_0' \wedge \mathsf{Inst}[F_3][0] \wedge \bigwedge R = (\texttt{len}(x_0) \leq 0) \wedge \texttt{len}(7) \leq 0 \wedge x_0 = 7$.

Dafny example. Our algorithm can synthesize various triggering terms that expose the unsoundness from Fig. 2, depending on the values of its parameters. We explain here one, obtained for $\sigma = 0.1$. For $\texttt{depth} = 0$, the algorithm checks each formula F_0 – F_4 in isolation. As they are all individually satisfiable, it continues with $\texttt{depth} = 1$. Due to space constraints, we only present the iteration for F_3.

F_3 shares at least two uninterpreted symbols with each of the other formulas, so there are various alternative rewritings: $s_3 = \texttt{Empty}(t_1)$, $s_3 = s_4$, $s_3 = \texttt{Build}(s_4, i_4, v_4, len_4)$, etc. As we consider clusters-rewritings pairs in which each quantified variable has maximum one rewriting, one such pair is $(C = \{F_4\}, R = \{s_3 = \texttt{Build}(s_4, i_4, v_4, len_4)\})$. F_4 has two disjuncts, so its instantiations are $\mathsf{Inst}[F_4] = \{\texttt{typ}(s_4) = \texttt{Type}(\texttt{typ}(v_4)),\ \neg(\texttt{type}(s_4) = \texttt{Type}(\texttt{typ}(v_4)) \wedge \texttt{Length}(\texttt{Build}(s_4, i_4, v_4, len_4)) = len_4\}$. From these instantiations and the rewritings R, we derive two formulas: $G_0 = \neg F_3' \wedge \mathsf{Inst}[F_4][0] \wedge \bigwedge R$, with the model $s_3 = s$, $s_4 = s'$, $i_4 = 0$, $v_4 = v$, $len_4 = 1$ and $G_1 = \neg F_3' \wedge \mathsf{Inst}[F_4][1] \wedge \bigwedge R$, with the model $s_3 = s, s_4 = s', i_4 = 0, v_4 = v, len_4 = -1$, where s, s', and v are fresh variables of type U (We use indexes for the G formulas to refer to them later). We then construct the candidate triggering terms from the patterns of the formulas F_3 and F_4. We replace s_3 by its right hand side in the rewriting, i.e., $\texttt{Build}(s_4, i_4, v_4, len_4)$, and all the other quantified variables by their constants from the model. The result after removing redundant terms is: $T_0 = \texttt{dummy}(\texttt{Length}(\texttt{Build}(t, 0, v, 1)))$ and $T_1 = \texttt{dummy}(\texttt{Length}(\texttt{Build}(t, 0, v, -1)))$.

Since the validation step fails for both T_0 and T_1, we continue with the other (C, R) pairs, the remaining quantified conjuncts and their similarity clusters.

If no candidate term is sufficient to prove unsat, our algorithm expends the clusters. To scale to real-world axiomatizations, it efficiently reuses the results from the previous iterations; i.e., it prunes the search space if a previously synthesized formula G is unsatisfiable and it strengthens G if it is satisfiable. The pair $(C = \{F_4\}, R = \{s_3 = \texttt{Build}(s_4, i_4, v_4, len_4)\})$ can be extended to $(C = \{F_4, F_1\}, R = \{s_3 = \texttt{Build}(s_4, i_4, v_4, len_4), s_4 = \texttt{Empty}(t_1), t_1 = \texttt{typ}(v_4)\})$, as F_1 is similar with F_4 through the rewritings $R = \{s_4 = \texttt{Empty}(t_1), t_1 = \texttt{typ}(v_4)\}$. We thus conjoin the instantiation of F_1 and the two additional rewritings to the formulas G_0 and G_1 from the previous iteration. This is equivalent to recomputing the similarity cluster, the rewritings, and the combinations of instantiations. We then obtain: $G_0' = G_0 \wedge (\texttt{type}(\texttt{Empty}(t_1)) = \texttt{Type}(t_1)) \wedge (s_4 = \texttt{Empty}(t_1)) \wedge (t_1 = \texttt{typ}(v_4))$, which is unsatisfiable, and $G_1' = G_1 \wedge (\texttt{type}(\texttt{Empty}(t_1)) = \texttt{Type}(t_1)) \wedge (s_4 = \texttt{Empty}(t_1)) \wedge (t_1 = \texttt{typ}(v_4))$ with the model: $s_3 = s$, $s_4 = s'$, $i_4 = 0$, $v_4 = v$, $len_4 = -1$, $t_1 = t$, where s, s', v, and t are fresh variables of types U and V. From this model and the rewritings we construct the triggering term $T = \texttt{dummy}\ (\texttt{Length}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)))$, which is sufficient to expose the inconsistency between F_3 and F_4.

$$F\colon \forall a\colon \text{Int}, b\colon \text{Int}, size\colon \text{Int} :: \{\texttt{both\_ptr}(a, b, size)\}$$
$$\texttt{both\_ptr}(a, b, size) * size \leq a - b$$

Fig. 15. Inconsistent formula from a VCC/HAVOC [33,11] benchmark from SMT-COMP [35], which cannot be proved unsat by MBQI. Our synthesized triggering term $\texttt{dummy}(\texttt{both\_ptr}(-2, -1, 0))$ allows E-matching to refute the formula.

VCC/HAVOC example. Fig. 15 presents a fragment of a benchmark which could be solved by our algorithm, but could not be proved by MBQI. F, which was part of a set of 160 formulas, is inconsistent by itself: when $size = 0$, E-matching can refute it for any integer values a, b, such that $a \leq b$. Our algorithm synthesizes the required triggering term in ≈7s because it initially considers each quantified conjunct in isolation. The formula $G = \neg F' = \texttt{both\_ptr}(a, b, size) * size > a - b$ is satisfiable and the simplest models the solver can provide (without assigning an interpretation to the uninterpreted function `both_ptr`) all include $size = 0$.

External patterns. For the examples discussed so far, the functions used as patterns were also present in the body of the quantifiers. However, to have a better control over the instantiations, one can also write formulas where the patterns are additional uninterpreted functions, which do not appear in the bodies. Such external patterns are not uncommon in proof obligations. Fig. 16 shows an example, which uses the synonym functions technique [21] to avoid matching loops. `sum` and `sum_syn` compute the sum of the elements of a sequence, between a

$$F_0 : \forall s_0 : \text{ISeq}, l_0 : \text{Int}, h_0 : \text{Int} :: \{\texttt{sum}(s_0, l_0, h_0)\}\ \texttt{sum}(s_0, l_0, h_0) = \texttt{sum\_syn}(s_0, l_0, h_0)$$
$$F_1 : \forall s_1 : \text{ISeq}, l_1 : \text{Int}, h_1 : \text{Int} :: \{\texttt{sum}(s_1, l_1, h_1)\}\ \neg(l_1 \geq h_1) \vee \texttt{sum\_syn}(s_1, l_1, h_1) = 0$$
$$F_2 : \forall s_2 : \text{ISeq}, l_2 : \text{Int}, h_2 : \text{Int} :: \{\texttt{sum}(s_2, l_2, h_2)\}\ \neg(l_2 \leq h_2) \vee$$
$$(\texttt{sum\_syn}(s_2, l_2, h_2) = \texttt{sum\_syn}(s_2, l_2 + 1, h_2) + \texttt{seq.nth}(s_2, l_2))$$
$$F_3 : \texttt{seq.nth}(\texttt{empty}, 0) = -1$$

Fig. 16. Formulas with synonym functions as patterns that axiomatize sequence comprehensions and set contradictory constraints on the function `sum_syn`. ISeq is a user-defined type, `empty` is a user-defined constant of type ISeq (i.e., the empty sequence).

lower and an upper bound. The two functions are identical (according to F_0), but only `sum` is used as a pattern. For equal bounds, F_1 and F_2 set contradictory constraints on the interpretation of `sum_syn`. `seq.nth` returns the n-th element of the sequence. Using the information from the quantifier-free formula F_3, our algorithm generates the triggering term $T = \texttt{dummy}(\texttt{sum}(\texttt{empty}, 0, 0), \texttt{sum}(\texttt{empty}, 0 + 1, 0))$. The term "0+1" comes from the rewriting $l_0 = l_2 + 1$. Addition is a built-in function, but is used as an argument to the uninterpreted function `sum_syn`, thus, it is supported by our unification. Our algorithm is syntactic, so it does not perform arithmetic operations, it just substitutes l_2 with its value from the model. The solver then performs theory reasoning and concludes unsat.

Multi-patterns and alternative patterns. SMT solvers allow patterns to contain multiple terms, all of which must be present to perform an instantiation. F_1 in Fig. 17 has such a *multi-pattern* and can be instantiated only when triggering terms that match both $\{\texttt{g}(b_1)\}$ *and* $\{\texttt{f}(x_1)\}$ are present in the SMT run. Our algorithm directly supports multi-patterns, as the procedure candidateTerm instantiates all the patterns from the given cluster (see Alg. 3, line 9). For the example from Fig. 17, our technique synthesizes the triggering term $T = \texttt{dummy}(\texttt{f}(7), \texttt{g}(b))$ from the rewritings $R = \{x_0 = x_1\}$ and the model of the formula $G = \neg F_0' \wedge \mathsf{Inst}[F_1][1] \wedge \bigwedge R = (\texttt{f}(x_0) = 7) \wedge (\neg \texttt{g}(b_1) \wedge \texttt{f}(x_1) = x_1) \wedge (x_0 = x_1)$. b is a fresh, unconstrained variable of the uninterpreted type B.

$$F_0 : \forall x_0 : \text{Int} :: \{\texttt{f}(x_0)\}\ \texttt{f}(x_0) \neq 7$$
$$F_1 : \forall b_1 : \text{B}, x_1 : \text{Int} :: \{\texttt{g}(b_1), \texttt{f}(x_1)\}\ \texttt{g}(b_1) \vee (\texttt{f}(x_1) = x_1)$$
$$F_2 : \forall b_2 : \text{B} :: \{\texttt{g}(b_2)\}\ \neg \texttt{g}(b_2)$$

Fig. 17. Formulas that set contradictory constraints on the function `f`. F_1 has a multi-pattern. B is an uninterpreted type.

Formulas can also contain *alternative patterns*. For example, the quantified formula $\forall x : \text{Int} :: \{\texttt{f}(x)\}\ \{\texttt{h}(x)\}\ \texttt{f}(x) \neq 7 \vee \texttt{h}(x) = 6$ is instantiated only if there

exists a triggering term that matches $\{\mathtt{f}(x)\}$ *or* one that matches $\{\mathtt{h}(x)\}$. Our algorithm does not differentiate between multi-patterns and alternative patterns, thus it always synthesizes the arguments for *all* the patterns of a cluster. For alternative patterns, this results in an over-approximation of the set of necessary triggering terms. However, the minimization step (performed before returning the triggering term that successfully validates), removes the unnecessary terms.

Appendix E Optimizations

In this section, we present various optimizations implemented in our tool, which allow the algorithm to scale to real-world verification benchmarks.

Grammar. The grammar from Fig. 6 allows us to simplify the presentation of the algorithm. However, eliminating conjunctions by applying distributivity and splitting (as described in Sec. 3.1) can result in an exponential increase in the number of terms and introduce redundancy, affecting the performance. Conjunction elimination is not implemented in Z3's `NNF` tact, thus it is not performed automatically. We apply this transformation only at the top-level, i.e., we do not recursively distribute disjunctions over conjunctions. For this reason, the input conjuncts F supported by our tool can actually contain conjunctions, in which case we use an extended algorithm when computing the instantiations, to ensue that all the resulting G formulas are still quantifier-free. The number of conjuncts and the number of quantifiers reported in Tab. 1 and Tab. 2 were computed *before* applying distributivity, thus they are not artificially increased.

Rewritings. The restrictive shapes of our rewritings (from Sec. 3.2, step 2), ensure that their number is finite, because if it exists, the most general unifier is unique up to variable renaming, i.e., substitutions of the type $\{x_i \to x_j, x_j \to x_i\}$ [6]. (Such substitutions are rewritings of the shapes (1) and (2) where rhs is also a quantified variable.) However, for most practical examples, the number of rewritings is very large, thus our implementation identifies them lazily, in increasing order of cardinality. If a rewriting $r \in R$ leads to an unsat G formula for some instantations, then we discard all the subsequent G formulas that contain r and the same instantiations (they will also be unsatisfiable). To make sure that the algorithm terminates within a given amount of time, in our experiments we bound the number of G formulas derived for each quantified conjunct F to 100.

Instantiations. Our implementation computes lazily the Cartesian product of the instantiations (Alg. 1, line 9), since it can also have a high number of elements. However, many of them are in practice unsatisfiable, thus our tool efficiently identifies trivial conflicts (e.g., $\neg D_i \wedge D_i$), pruning the search space accordingly.

Candidate terms. To improve the performance of our algorithm, we keep track of all the candidate triggering terms that failed to validate (i.e., of the models from which they were synthesized). Then, we add constraints (similar to the conjunct $\neg$`model` from Alg. 1, line 19) to ensure the solver does not provide previously-seen models for the quantified variables from the same set of patterns.

Appendix F Limitations

Next, we discuss the limitations of our technique, as well as possible solutions.

Applicability. Our algorithm effectively addresses the prevalent cause of failed unsatisfiability proofs in verification, i.e., missing triggering terms. However, it is not able to remedy incompleteness in the solver's decision procedures caused by undecidable theories. Also, our technique is tailored to *unsatisfiability* proofs, for which it is sufficient to find *one* instantiation of the quantifiers that leads to an inconsistency. In contrast, sat proofs need to consider *all* instantiations, which cannot be solved by triggering; they cannot be reduced to unsat proofs by negating the input, as the negation cannot be encoded in SMT (see Sec. 2).

SMT solvers. Our algorithm can synthesize triggering terms as long as the SMT solver can find models for our quantifier-free formulas. However, solvers are incomplete, i.e., they can return *unknown* and generate only *partial models*, which are not guaranteed to be correct. Nonetheless, we also use partial models, as the validation step (step 4 in Fig. 4) ensures that they do not lead to false positives.

Patterns. Since our algorithm is based on patterns (provided or inferred), it will not succeed if they do not permit the necessary instantiations. For example, the formula $\forall x\colon \mathrm{Int}, y\colon \mathrm{Int} :: x = y$ is unsatisfiable. However, the SMT solver cannot automatically infer a pattern from the body of the quantifier, since equality is an interpreted function and must not be used as a pattern. Thus E-matching (and implicitly our algorithm) cannot solve this example, unless the user provides as pattern some uninterpreted function that mentions both x and y (e.g., $\mathtt{f}(x, y)$).

Bounds and rewritings. Synthesizing triggering terms is generally undecidable. We ensure termination by bounding the search space through various customizable parameters, thus our algorithm misses results not found within these bounds. We also only unify applications of uninterpreted functions, which are common in verification. Efficiently supporting interpreted functions (especially equality) is very challenging for inputs with a small number of types (e.g., from Boogie [7]).

Despite these limitations, our algorithm effectively synthesizes the triggering terms required in practical examples, as we have experimentally shown in Sec. 4.

Appendix G SMT-COMP Benchmarks Selection

Next, we describe our pre-filtering step, for identifying files with triggering issues.

We collected 27,716 benchmarks from SMT-COMP 2020 [35] (single query track), with ground truth *unsat* and at least one pattern (as this suggests they were designed for E-matching). We then ran Z3 to infer the missing patterns and to transform the formulas into NNF and removed all benchmarks for which the inference or the transformation did not succeed within 600s per file and

4s per formula. We also removed the files with features not yet supported by PySMT [15], the parsing library used in our experiments (e.g., sort signatures in datatypes declarations, but we did extend PySMT to handle, e.g., patterns and overloaded functions). This filtering resulted in 6,481 benchmarks. We then ran E-matching and kept only those examples that could not be solved within 600s due to incompleteness in instantiating quantifiers (our work only targets this incompleteness, but the SMT-COMP suite also contains other solving challenges). We thus obtained 61 files from Spec# [8], VCC [33], Havoc [11], Simplify [13], and the Bit-Width-Independent (BWI) encoding [27], summarized in Tab. 2.