Identifying Overly Restrictive Matching Patterns in SMT-Based Program Verifiers

Alexandra Bugariu$^{(\boxtimes)}$, Arshavir Ter-Gabrielyan, and Peter Müller

Department of Computer Science, ETH Zurich, Zürich, 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.

Keywords: Matching patterns · Triggering terms · SMT · E-matching

1 Introduction

Proof obligations frequently contain universal quantifiers, both in the specification and to encode the semantics of the programming language. Most deductive verifiers [4,5,8,12,15,19,36] rely on SMT solvers to discharge the proof obligations via E-matching [14]. 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 $\forall x\colon \text{Int}, y\colon \text{Int} :: \{\mathtt{f}(x,y)\}\ (x = y) \wedge \neg\mathtt{f}(x,y)$ 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 [20,23]. 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 [14]). 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. Figure 1 illustrates this

```
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) == 1) &&
        (nxt(x) != x ==> len(x) == len(nxt(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.

case. 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 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. However, realistic proof obligations often contain hundreds of quantifiers [33], 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 those 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 [13,21,30].

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 [19] 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 l_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.

This Work. For SMT-based deductive verifiers, discharging proof obligations and revealing inconsistencies in axiomatizations require a solver to prove unsat

$F_0 : \forall t_0 : V :: \{\texttt{Type}(t_0)\}\ t_0 = \texttt{ElemType}(\texttt{Type}(t_0))$

$F_1 : \forall t_1 : V :: \{\texttt{Empty}(t_1)\}\ \texttt{typ}(\texttt{Empty}(t_1)) = \texttt{Type}(t_1)$

$F_2 : \forall s_2 : U, i_2 : \text{Int}, v_2 : U, l_2 : \text{Int} :: \{\texttt{Build}(s_2, i_2, v_2, l_2)\}$
$\quad\quad \texttt{typ}(\texttt{Build}(s_2, i_2, v_2, l_2)) = \texttt{Type}(\texttt{typ}(v_2))$

$F_3 : \forall s_3 : U :: \{\texttt{Len}(s_3)\}\ \neg(\texttt{typ}(s_3) = \texttt{Type}(\texttt{ElemType}(\texttt{typ}(s_3)))) \vee (0 \leq \texttt{Len}(s_3))$

$F_4 : \forall s_4 : U, i_4 : \text{Int}, v_4 : U, l_4 : \text{Int} :: \{\texttt{Len}(\texttt{Build}(s_4, i_4, v_4, l_4))\}$
$\quad\quad \neg(\texttt{typ}(s_4) = \texttt{Type}(\texttt{typ}(v_4))) \vee (\texttt{Len}(\texttt{Build}(s_4, i_4, v_4, l_4)) = l_4)$

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].

via E-matching. (Verification techniques based on proof assistants are out of scope.) 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 [17,20].

Fixing the Unsoundness. In Fig. 2, our triggering term `Len(Build(Empty` `(typ(v)), 0, v, −1))` (for a fresh value v) is sufficient to detect the unsoundness (as shown in Appx. A of [11]). 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. Figure 3 illustrates another scenario: Boogie's [7] map axiomatization is inconsistent by design at the SMT level [22], but this behavior cannot be exposed from Boogie, as the type system prevents the required instantiations. Thus 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. They could *unsoundly* classify an incorrect program that uses this map axiomatization as correct. Since F_2 states

$F_0 \colon \forall kt_0 \colon \mathrm{V}, vt_0 \colon \mathrm{V} :: \{\mathtt{Type}(kt_0, vt_0)\}\ \mathtt{ValTypeInv}(\mathtt{Type}(kt_0, vt_0)) = vt_0$

$F_1 \colon \forall m_1 \colon \mathrm{U}, k_1 \colon \mathrm{U}, v_1 \colon \mathrm{U} :: \{\mathtt{Select}(m_1, k_1, v_1)\}$
$\qquad\quad \mathtt{typ}(\mathtt{Select}(m_1, k_1, v_1)) = \mathtt{ValTypeInv}(\mathtt{typ}(m_1))$

$F_2 \colon \forall m_2 \colon \mathrm{U}, k_2 \colon \mathrm{U}, x_2 \colon \mathrm{U}, v_2 \colon \mathrm{U} :: \{\mathtt{Store}(m_2, k_2, x_2, v_2)\}$
$\qquad\quad \mathtt{typ}(\mathtt{Store}(m_2, k_2, x_2, v_2)) = \mathtt{Type}(\mathtt{typ}(k_2), \mathtt{typ}(v_2))$

$F_3 \colon \forall m_3 \colon \mathrm{U}, k_3 \colon \mathrm{U}, x_3 \colon \mathrm{U}, v_3 \colon \mathrm{U}, k_3' \colon \mathrm{U}, v_3' \colon \mathrm{U} :: \{\mathtt{Select}(\mathtt{Store}(m_3, k_3, x_3, v_3), k_3', v_3')\}$
$\qquad\quad (k_3 = k_3') \vee (\mathtt{Select}(\mathtt{Store}(m_3, k_3, x_3, v_3), k_3', v_3') = \mathtt{Select}(m_3, k_3', v_3'))$

Fig. 3. Fragment of Boogie's map axiomatization, which is sound only modulo patterns. U and V are uninterpreted types. All the named functions are uninterpreted.

that storing a key-value pair into a map results in a new map with a potentially *different* type, one can prove that two *different* types (e.g., Boolean and Int) are equal in SMT. This example 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 [4,5,8,12,15,19,36], and may produce unsound results for verifiers designed for E-matching with axiomatizations sound only modulo patterns.

Contributions. This paper makes the following technical contributions:

1. We present the first automated technique that allows the developers to detect *completeness* issues in program verifiers and *soundness* problems in their axiomatizations. Moreover, our approach helps them devise better triggering strategies for *all* future runs of their tool with E-matching.
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 65,6% 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: Sect. 2 gives an overview of our technique; the details follow in Sect. 3. In Sect. 4, we present our experimental results. We discuss related work in Sect. 5, and conclude in Sect. 6. Extensions of our algorithm, optimizations, more details about E-matching and the evaluation, and additional examples can be found in the extended version of our paper [11].

2   Overview

Our goal is to synthesize missing triggering terms, i.e., concrete instantiations for (a small subset of) the quantified variables of an input formula I, which are
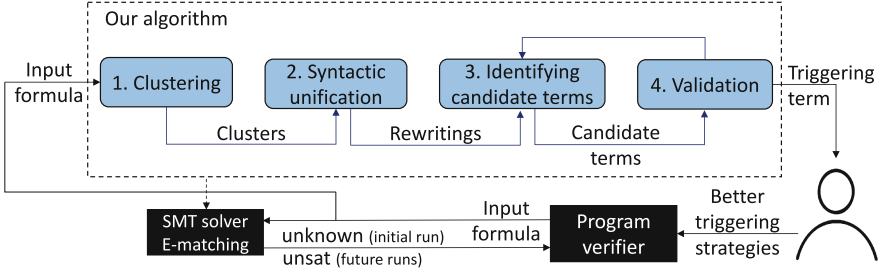
Fig. 4. Main steps of our algorithm that helps the developers of program verifiers devise better triggering strategies. Rounded boxes depict processing steps and arrows data.

necessary for the solver to prove its unsatisfiablity. Intuitively, these triggering terms include *counter-examples* to the satisfiability of I and can be obtained from a model of its negation. For example, $I = \forall n \colon \mathrm{Int} :: n > 7$ is unsatisfiable, and a counter-example $n = 6$ is a model of its negation $\neg I = \exists n \colon \mathrm{Int} :: n \leq 7$.

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

Let F be a second-order formula. We define its *approximation* as:

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

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

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

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 $I = F_0 \wedge F_1 \wedge \ldots$ is unsatisfiable, the solver requires the triggering term $\mathtt{f(g(7))}$. The corresponding instantiations of F_0 and F_1 generate contradictory constraints: $\mathtt{f(g(7))} \neq 7$ and $\mathtt{f(g(7))} = 7$. In the following, we explain how we obtain this triggering term systematically.

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

Fig. 5. Formulas that set contradictory constraints on the function f. Synthesizing the triggering term $\mathtt{f(g(7))}$ 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 its *quantified* conjuncts F 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 directly.) We 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 $g(x_1)$ 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 $f(g(x_1))$ if we perform the rewritings $R = \{x_0 = g(x_1)\}$.

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 other formula from C.

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:

$$F_{Inst} = F[\exists \overline{\mathbf{x}} \,/\, \forall \overline{\mathbf{x}}] \tag{**}$$

where $\overline{\mathbf{x}}$ are variables. Then $G = \neg F_{0\approx} \wedge F_{1\,Inst} \wedge \bigwedge R$ is equivalent to $(f(x_0) = 7) \wedge (f(g(x_1)) = x_1) \wedge (x_0 = g(x_1))$. (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 $g(x_1)$ (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 $f(g(7))$.

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

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

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.

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 $\texttt{f}(\texttt{g}(7))$ is sufficient to complete the proof.

3   Synthesizing Triggering Terms

Next, we define the input formulas (Sect. 3.1), explain the details of our algorithm (Sect. 3.2) and discuss its limitations (Sect. 3.3). Appx. C and Appx. E of [11] present extensions that enable complex proofs and optimizations used in Sect. 4.

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 of [11] addresses 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*.

3.2   Algorithm

The pseudo-code of our algorithm is given in Algorithm 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 later in this section.

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

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 $\mid F$ is $\forall \overline{x} :: F'$ **do**

4   **foreach** $(C, R) \in$ clustersRewritings(I, F, σ, depth) **do**   // Steps 1, 2

5   Inst $\longleftarrow \{\}$

6   **foreach** $f \in C \mid f$ is $\forall \overline{x} :: D_0 \vee \ldots \vee D_n$ or $D_0 \vee \ldots \vee D_n$ **do**

7   Inst$[f] \longleftarrow \{(\bigwedge_{0 \leq j < k} \neg D_j) \wedge D_k \mid 0 \leq k \leq n\}$

8   Inst$[F] \longleftarrow \{\neg F'\}$

9   **foreach** $H \in \bigtimes \{$Inst$[f] \mid f \in \{F\} \cup C\}$ **do**   // Cartesian product

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   // Avoid this model next iteration

20   **return** None

The algorithm iterates over each *quantified* conjunct F of I (Algorithm 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:

Step 1: Clustering. It constructs clusters of formulas similar to F (Algorithm 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, $S_0 = \{$f$\}$, $S_1 = \{$f, g$\}$, $J(F_0, F_1) = \frac{|\{f\}|}{|\{f,g\}|} = 0.5$.

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

Algorithm 2: Auxiliary procedure for Algorithm 1, which identifies clusters of formulas similar to 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 restricted shapes, defined in text (step 2).

Arguments : `I` — input formula, also treated as set of conjuncts
F — quantified conjunct of `I`, i.e., $F \in I \mid F$ is $\forall \overline{x} :: F'$
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   simFormulas $\longleftarrow \{f \mid f \in I \setminus \{F\}$ and $\mathtt{sim}_I^{\mathtt{depth}}(F, f, \sigma)\}$ `// Step 1`
6   rewritings $\longleftarrow \{\}$
7   **foreach** $f \in$ simFormulas **do**
8   rws $\longleftarrow$ unify(F, f) `// Step 2`
9   **if** rws $= \varnothing$ and (f is $\forall \overline{x} :: D_0 \vee \ldots \vee D_n$) **then**
10   simFormulas $\longleftarrow$ simFormulas $\setminus \{f\}$
11   rewritings$[f] \longleftarrow$ rws
12   **return** $\{(C, R) \mid C \subseteq$ simFormulas and $(\forall r \in R, \exists f \in C : r \in$ rewritings$[f])$
            and $(\forall x \in \mathtt{qvars}(C): |\{r \mid r \in R$ and $x = \mathtt{lhs}(r)\}| \leq 1)\}$

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

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. Next (Algorithm 2, line 8) we identify *rewritings*, i.e., constraints under which two similar *quantified* formulas share terms. (Appx. D of [11] presents the quantifier-free case.) 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 `f` they have one of the following two 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 for the restricted 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 Algorithm 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 $\{(\{F_1\}, \{x_0 = x_1\}), (\{F_2\}, \{x_0 = x_2\})\}$.

Step 3: Identifying Candidate Terms. From the clusters and the rewritings (identified before), we then derive *quantifier-free* formulas G (Algorithm 1, line 10), and, if they are satisfiable, construct the candidate triggering terms from their models (Algorithm 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 Algorithm 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 each similar formula, of the form: $(\bigwedge_{0 \leq j < k} \neg D_j) \wedge D_k, \forall k: 0 \leq k \leq n$ (see Algorithm 1, line 7). To consider all the combinations of disjuncts, we derive the formula G from the Cartesian product of the instantiations (Algorithm 1, line 9). (To present the pseudo-code in a concise way, we store $\neg F'$ in the instantiations map as well (Algorithm 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: $\mathsf{Inst}[F_1] = \{x_1 \geq 1, (x_1 < 1) \wedge (\mathtt{f}(x_1) = 6)\}$. The formula $G = (x_0 > -1) \wedge (\mathtt{f}(x_0) \leq 7) \wedge (x_1 \geq 1) \wedge (x_0 = x_1)$ for the first instantiation is satisfiable, but none of the values the solver can assign to x_0

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

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.

Algorithm 3: Auxiliary procedure for Algorithm 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
$\qquad\qquad\quad$ R — set of rewritings for the cluster
$\qquad\qquad\quad$ `model` — SMT model, mapping variables to values
Result: A triggering term with no semantic information

---

1 **Procedure** candidateTerm
2 $\quad$ $P_0, \ldots, P_k \longleftarrow$ `patterns`(C)
3 $\quad$ **while** $R \neq \varnothing$ **do**
4 $\quad\quad$ choose and remove $r \longleftarrow (x = rhs)$ from R
5 $\quad\quad$ $P_0, \ldots, P_k \longleftarrow (P_0, \ldots, P_k)[\, rhs/x\,]$
6 $\quad\quad$ $R \longleftarrow R\,[\, rhs/x\,]$
7 $\quad$ **foreach** $x \in$ `qvars`(C) **do**
8 $\quad\quad$ $P_0, \ldots, P_k \longleftarrow (P_0, \ldots, P_k)[\, \texttt{model}(x)/x\,]$
9 $\quad$ **return** `"dummy"` + `"("` + $P_0, \ldots, P_k$ + `")"`

(which are all greater or equal to 1) are sufficient for the unsatisfiability proof to succeed. The second instantiation adds additional constraints: instead of $x_1 \geq 1$, it requires $(x_1 < 1) \wedge (\texttt{f}(x_1) = 6)$. The resulting G formula has a unique solution for x_0, namely 0, and the triggering term $\texttt{f}(0)$ is sufficient to prove unsat.

$\qquad$ The procedure candidateTerm from Algorithm 3 synthesizes a candidate triggering term T from the model of G and the rewritings R. We first collect all the patterns of the formulas from the cluster C (Algorithm 3, line 2), i.e., of F and of its similar conjuncts (see Algorithm 1, line 15). Then, we *apply* the rewritings, in an arbitrary order (Algorithm 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 (Algorithm 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.

$$F_0 : \forall x_0 : \text{Int} :: \{\texttt{f}(x_0)\} \; \neg(x_0 > -1) \vee (\texttt{f}(x_0) > 7)$$
$$F_1 : \forall x_1 : \text{Int} :: \{\texttt{f}(x_1)\} \; \neg(x_1 < 1) \vee (\texttt{f}(x_1) = 6)$$

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

Step 4: Validation. We validate the candidate triggering term T by checking if $\mathbf{I} \wedge T$ is unsatisfiable, i.e., if these particular interpretations for the uninterpreted functions generalize to all interpretations (Algorithm 1, line 16). If this is the case then we return the *minimized* triggering term (Algorithm 1, line 18). The `dummy` function has multiple arguments, each of them corresponding to one pattern from the cluster (Algorithm 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 (Algorithm 1, lines 19 and 11). Appx. B of [11] discusses heuristics for obtaining *diverse models*.

Nested Quantifiers. Our algorithm also supports nested quantifiers. Nested existential quantifiers in positive positions and nested universal quantifiers in negative positions are replaced in NNF by new, uninterpreted Skolem functions. Step 2 is also applicable to them: Skolem functions with arguments (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, the result is imprecise: the unification of $\mathbf{f}(x_0, \mathtt{skolem}())$ and $\mathbf{f}(x_1, 1)$ 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), we extend the algorithm to *recursively instantiate* similar formulas with nested quantifiers when computing the instantiations.

3.3   Limitations

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

Applicability. Our algorithm effectively addresses a common cause of failed unsatisfiability proofs in program verification, i.e., missing triggering terms. Other causes (e.g., incompleteness in the solver's decision procedures due to undecidable theories) are beyond the scope of our work. Also, our algorithm is tailored to *unsatisfiability* proofs; satisfiability proofs cannot be reduced to unsatisfiability proofs by negating the input, because the negation cannot usually be encoded in SMT (as we have illustrated in Sect. 2).

SMT Solvers. Our algorithm synthesizes 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 occur in 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 experimentally show next.

4   Evaluation

Evaluating our work requires benchmarks with known triggering issues (i.e., for which E-matching yields *unknown*). Since there is no publicly available suite, in Sect. 4.1 we used manually-collected benchmarks from four verifiers [19,25,35,38]. Our algorithm succeeded for 65,6%. To evaluate its applicability to other verifiers, in Sect. 4.2 we used SMT-COMP [33] inputs. As they were not designed to expose triggering issues, we developed a filtering step (see Appx. F of [11]) to automatically identify the subset that falls into this category. The results show that our algorithm is suited also for [8,12,32]. Section 4.3 illustrates that our triggering terms are simpler than the unsat proofs produced by quantifier instantiation and refutation techniques, enabling one to fix the root cause of the revealed issues.

Setup. We used Z3 (4.8.10) [24] 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 [19], F* [35], Gobra [38], and Viper [25]. We reconstructed 4, respectively 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 Table 1, columns 3–4.

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 [28]), and Vampire [18] (in CASC mode [34], using Z3 for ground theory reasoning).

Source	#	#F min-max	#$\forall$ min-max	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		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	**32**					21 (65,6%)				19 (59,3%)	0 (0%)	30 (93,7%)

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

Configurations. We ran our tool with five configurations, to also analyze the impact of its parameters (see Algorithm 1 and Appx. C of [11]). The default configuration C0 has: $\sigma = 0.3$ (similarity threshold), $\beta = 64$ (batch size, i.e., the number of candidate terms validated together), $\neg$type (no type-based constraints), $\neg$sub (no unification for sub-terms). The other configurations differ from C0 in the parameters shown in Table 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 Table 1 show the number of files solved by each configuration, column 10 summarizes the files solved by at least one. Overall, we found suited triggering terms for 65,6%, including all F* and Gobra benchmarks. An F* unsoundness exposed by all configurations in ≈60s is given in [11] (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 [11] (Fig. 11). Gobra's team spent one week to identify 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`. Both result in E-matching refuting the proof obligation in under 0.1s. We also exposed another case where Boogie (used by Viper) is sound only modulo patterns (as in Fig. 3).

4.2   Effectiveness on SMT-COMP Benchmarks

Next, we considered 61 SMT-COMP [33] benchmarks from Spec# [8], VCC [32], Havoc [12], Simplify [14], and the Bit-Width-Independent (BWI) encoding [26].

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

Source	#	#F min-max	#∀ min-max	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	61					29 (47,5%)				41 (67,2%)	0 (0%)	55 (90,1%)

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

Results. The results are shown in Table 2. Our algorithm enabled E-matching to refute 47.5% 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 time-out. 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 work, tool developers could try to *manually* identify triggering issues from refutation proofs, but these do not consider patterns and are harder to understand. Columns 11–13 in Table 1 and Table 2 show the number of proofs produced by Z3 with MBQI [16], CVC4 [9] with enumerative instantiation [28], and Vampire [18] using Z3 for ground theory reasoning [27] and the CASC [34] 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 our configurations solved two VCC/Havoc files not solved by MBQI (Appx. D of [11] shows an example). Moreover, our triggering terms are much simpler and directly highlight the root cause of the issues. Compared to our generated term `loc(a,2)`, MBQI's proof for Viper's array maximum example has 2135 lines and over 700 reasoning steps, while Vampire's proof has 348 lines and 101 inference steps. Other proofs have similar complexity.

Vampire and MBQI cannot replace our technique: as most deductive verifiers employ E-matching, it is important to help the developers use the algorithm of their choice and return sound results even if they rely on patterns for soundness (as in Fig. 3). Our tool can also produce multiple triggering terms (see Appx. C of [11]), thus it can reveal *multiple* triggering issues for the same input formula.

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* [29] 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 [31] 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* [28] 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 Sect. 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 [18]) also generate refutation proofs. More recent works combine a superposition calculus with theory reasoning [27,37], 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.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
8. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011)
9. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
10. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
11. Bugariu, A., Ter-Gabrielyan, A., Müller, P.: Identifying overly restrictive matching patterns in SMT-based program verifiers (extended version). Technical report, 2105.04385, arXiv (2021)
12. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_4
13. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 336–351. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71289-3_26
14. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005). https://doi.org/10.1145/1066100.1066102
15. Eilers, M., Müller, P.: Nagini: a static verifier for Python. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 596–603. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_33
16. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25

17. 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.) ECOOP 2013. LNCS, vol. 7920, pp. 451–476. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_19

18. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1

19. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20

20. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009, pp. 615–622. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1529282.1529411

21. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 307–321. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_24

22. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_26

23. Moskal, M.: Programming with triggers. In: SMT. ACM International Conference Proceeding Series, vol. 375, pp. 20–29. ACM (2009)

24. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

25. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2

26. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards bit-width-independent proofs in SMT solvers. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 366–384. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_22

27. 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

28. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_7

29. 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.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_12

30. Rudich, A., Darvas, Á., Müller, P.: Checking well-formedness of pure-method specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 68–83. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_7

31. Rümmer, P.: E-matching with free variables. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 359–374. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_28

32. 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/

33. SMT-COMP 2020: The 15th international satisfiability modulo theories competition (2020). https://smt-comp.github.io/2020/

34. Sutcliffe, G.: The CADE ATP system competition - CASC. AI Mag. **37**(2), 99–101 (2016)

35. Swamy, N., et al.: Dependent types and multi-monadic effects in F*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 256–270. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2837614.2837655

36. 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, PLDI 2013, pp. 387–398 (2013). https://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/

37. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_46

38. Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: modular specification and verification of Go programs. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 367–379. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_17