Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers (Extended Version)

ALEXANDRA BUGARIU, Department of Computer Science, ETH Zurich
ARSHAVIR TER-GABRIELYAN, DFINITY
PETER MÜLLER, Department of Computer Science, ETH Zurich

Universal quantifiers occur frequently in proof obligations produced by program verifiers, for instance, to axiomatize uninterpreted functions and to statically 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 proof are not instantiated; they may also conceal unsoundness caused by inconsistent axiomatizations. In this article, we present the first technique that identifies and helps the users and the developers of program verifiers remedy the effects of overly restrictive matching patterns. We designed a novel algorithm to synthesize missing triggering terms required to complete unsatisfiability proofs via E-matching. Tool developers can use this information to refine their matching patterns and prevent similar verification errors, or to fix a detected unsoundness.

CCS Concepts: • **Software and its engineering → Formal software verification;**

Additional Key Words and Phrases: Matching patterns, Triggering terms, SMT, E-matching

ACM Reference format:
Alexandra Bugariu, Arshavir Ter-Gabrielyan, and Peter Müller. 2023. Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers (Extended Version). *Form. Asp. Comput.* 35, 2, Article 7 (June 2023), 27 pages.
https://doi.org/10.1145/3571748

1 INTRODUCTION

Proof obligations frequently contain universal quantifiers, both in the specification and to encode the semantics of the programming language. Most deductive verifiers [10, 13, 16, 22, 26, 32, 50] rely on SMT solvers to discharge their proof obligations via *E-matching* [25]. This SMT algorithm requires syntactic matching patterns of ground terms (called *patterns* in the following), to control the instantiations of the quantifiers. For example, the pattern $\{\mathsf{f}(x, y)\}$ in the formula $\forall x\colon \mathrm{Int}, y\colon \mathrm{Int} :: \{\mathsf{f}(x, y)\}\ (x = y) \wedge \neg \mathsf{f}(x, y)$ instructs the solver to instantiate the quantifier *only* when it finds a *triggering term* that matches the pattern, e.g., $\mathsf{f}(7, z)$, where f is an *uninterpreted* function and z is a free integer variable. The patterns can be written manually or inferred automatically by the

Authors' addresses: A. Bugariu and P. Müller, Department of Computer Science, ETH Zurich, Universitätstrasse 6, 8092 Zürich, Switzerland; emails: {alexandra.bugariu, peter.mueller}@inf.ethz.ch; A. Ter-Gabrielyan, DFINITY Foundation, Genferstrasse 21, 8002 Zürich, Switzerland; email: arshavir.ter.gabrielyan@dfinity.org.

0934-5043/2023/06-ART7 $15.00
https://doi.org/10.1145/3571748

```
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 (written in Boogie [15]) that leads to a spurious verification error. The assertion follows from the axiom, but the axiom does not get instantiated without the triggering term `len(nxt(7))`.

solver or the verifier. However, devising them is challenging [33, 37]. 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 [25]). 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 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, the last node points to itself, and any added node increases the length of the list by one. The assertion directly follows from the axiom, yet the proof fails, as the proof obligation generated by the verifier for the `assert` statement does not contain any triggering term that matches the pattern `{len(nxt(x))}`. Thus, the axiom does not get instantiated. However, realistic proof obligations often contain *hundreds* of quantifiers [5], making 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 the proof obligations hold trivially. Consistency can be proved once and for all by showing the existence of a model that satisfies all the axioms, as part of the soundness proof of the verification technique. However, this solution is difficult to apply for those verifiers that 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 [23, 34, 45].

A more practical approach is to check if the axioms generated for a *given program* are consistent. However, this check also depends on triggering: the 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 [32] sequence axiomatization from June 2008 contained an inconsistency found only over a year later. A fragment of this axiomatization is shown in Figure 2.

The types U and V are uninterpreted. All the named functions are uninterpreted and are used to describe operations over *generic* sequences (their original names have been simplified for presentation purposes): `Type`: $\mathrm{V} \rightarrow \mathrm{V}$ represents the sequence's type, while `ElemType`: $\mathrm{V} \rightarrow \mathrm{V}$ denotes the type of the sequence's elements. Therefore, F_0 states that the elements of a sequence of t_0 (e.g., integers) have type t_0. The function `typ`: $\mathrm{U} \rightarrow \mathrm{V}$ returns the type of its argument, i.e., of the sequence (e.g., s_4 in F_4), or of its elements (e.g., v_4 in F_4). The elements of a sequence can also be sequences. `Empty`: $\mathrm{V} \rightarrow \mathrm{U}$ denotes an empty sequence of elements of a given type. `Build`: $\mathrm{U} \times \mathrm{Int} \times \mathrm{U} \times \mathrm{Int} \rightarrow \mathrm{U}$ creates a new sequence from the one provided as the first argument.

$$
\begin{aligned}
F_0: &\quad \forall t_0 : \mathrm{V} :: \{\texttt{Type}(t_0)\}\ t_0 = \texttt{ElemType}(\texttt{Type}(t_0))\\
F_1: &\quad \forall t_1 : \mathrm{V} :: \{\texttt{Empty}(t_1)\}\ \texttt{typ}(\texttt{Empty}(t_1)) = \texttt{Type}(t_1)\\
F_2: &\quad \forall s_2 : \mathrm{U}, i_2 : \mathrm{Int}, v_2 : \mathrm{U}, l_2 : \mathrm{Int} :: \{\texttt{Build}(s_2, i_2, v_2, l_2)\}\\
&\qquad\quad \texttt{typ}(\texttt{Build}(s_2, i_2, v_2, l_2)) = \texttt{Type}(\texttt{typ}(v_2))\\
F_3: &\quad \forall s_3 : \mathrm{U} :: \{\texttt{Len}(s_3)\}\ \neg(\texttt{typ}(s_3) = \texttt{Type}(\texttt{ElemType}(\texttt{typ}(s_3)))) \vee (0 \leq \texttt{Len}(s_3))\\
F_4: &\quad \forall s_4 : \mathrm{U}, i_4 : \mathrm{Int}, v_4 : \mathrm{U}, l_4 : \mathrm{Int} :: \{\texttt{Len}(\texttt{Build}(s_4, i_4, v_4, l_4))\}\\
&\qquad\quad \neg(\texttt{typ}(s_4) = \texttt{Type}(\texttt{typ}(v_4))) \vee (\texttt{Len}(\texttt{Build}(s_4, i_4, v_4, l_4)) = l_4)
\end{aligned}
$$

Fig. 2. Fragment of an old version of Dafny's [32] sequence axiomatization. U and V are uninterpreted types. All the named functions are uninterpreted. To improve readability, we use mathematical notation throughout this article instead of SMT-LIB syntax [18].

The axioms from Figure 2 express that sequences (including 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 unsoundness cannot be detected by checking the satisfiability of the formula $F_0 \wedge \ldots \wedge F_4$ because the axioms F_0–F_4 do not get instantiated.

This work. For SMT-based deductive verifiers, discharging proof obligations and revealing inconsistencies in axiomatizations require the SMT solver to prove *unsat* 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 unsatisfiability proof. These terms enable tool users and developers to understand and remedy the revealed completeness or soundness issue. Since the SMT encodings of different input programs and their specifications typically share axiomatizations or parts of the verification condition that encode the semantics of the programming language, fixing such issues benefits the verification of many or even all future runs of the verifier.

Fixing the incompleteness. For Figure 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 triggering term to their program. For example, adding the lines **`var`** `t:` **`int`**`; t := len(nxt(7))` before the assertion has no effect on the execution of the program but triggers the instantiation of the axiom. Tool *developers* can devise less restrictive patterns; e.g., 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, they can use this information to adapt the encoding to emit additional triggering terms enforcing certain instantiations [29, 33].

Fixing the unsoundness. In Figure 2, our synthesized triggering term $\texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0,\ v, -1))$ (for a fresh value v) is sufficient to detect the unsoundness (see Section 2). Tool *users* can use this triggering term to report bugs in the implementation of the program verifier, while tool *developers* can add an antecedent to F_4, which prevents the construction of sequences with negative lengths.

Soundness modulo patterns. Figure 3 illustrates another scenario: Boogie's [15] map axiomatization is inconsistent by design at the SMT level [35]; since F_2 states 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

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

Fig. 3. Fragment of Boogie's [15] map axiomatization, sound only modulo patterns. U and V are uninterpreted types. All the named functions are uninterpreted.

(e.g., Boolean and Int) are equal in SMT. However, this behavior cannot be exposed from Boogie, as the type system prevents the required instantiations. Thus, it does not affect Boogie's soundness.

Still, it is necessary to detect such cases as they could surface while using Boogie with quantifier instantiation strategies not based on E-matching (such as MBQI [28]) or with first-order provers (e.g., Vampire [31]), which *do not consider patterns*. They could thus *unsoundly* classify an *in*valid Boogie program that uses this map axiomatization as valid. Since the verifier proves the validity of the verification condition by showing that its negation is unsatisfiable, if the refutation algorithm yields *unsat*, the verifier concludes that the program fulfills its specification. This is the case when checking the axioms from Figure 3 with MBQI: the formula $F_0 \land \ldots \land F_3$ is equivalent to `false`, so *any* (even invalid) Boogie program whose SMT encoding contains the axioms F_0–F_3 is reported as valid.

This example shows that the problems tackled in our work cannot be solved simply by switching to alternative instantiation strategies, which ignore the patterns. First, these are not the preferred choices of most modern verifiers [10, 13, 16, 22, 26, 32, 50], and are unlikely to outperform E-matching. Second, these alternatives may produce unsound results for those verifiers designed for E-matching, with axiomatizations that are sound *only modulo patterns* (as the one from Figure 3).

Contributions. This article makes the following technical contributions:

(1) We present the first automated technique that allows users and developers of SMT-based program verifiers 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 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. Section 2 presents background information on E-matching. Section 3 gives an overview of our technique; the details follow in Section 4. In Section 5, we present our experimental results, in Section 6, we describe various optimizations that allow our algorithm to scale to real-world inputs, and in Section 7, we explain its limitations. We discuss related work in Section 8 and conclude in Section 9.

The current article is an extended and revised version of our paper "Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers" presented at FM'21 [20]. Compared to the conference paper, this article explains in more details the concept of "soundness modulo patterns" (Sections 1 and 5.3), describes how E-matching proves the unsatisfiability of an input formula (Section 2), presents five extensions of our algorithm (Section 4.3), and illustrates the (extended) algorithm on various examples: the Boogie and the Dafny examples from Figure 1 and Figure 2, as well as on a new VCC/Havoc [22, 47] benchmark and a list axiomatization with nested quantifiers (Section 4.4). Moreover, the current article explains how our algorithm supports quantifier-free formulas and more complex inputs with synonym functions as patterns, multi-patterns, and alternative patterns (Section 4.4). It also discusses the impact of various configurations of our technique on its effectiveness (Section 5.1), provides a mechanism for automatically selecting benchmarks with triggering issues for the evaluation (Section 5.2), includes a more detailed discussion about the differences between our algorithm and MBQI and Vampire (Section 5.3), presents threats to the validity of our experiments (Section 5.4), describes four optimizations implemented in our tool (Section 6), and discusses additional related work (Section 8) and various research directions we would like to explore in the future (Section 9).

2 BACKGROUND: E-MATCHING

In this section, we present the E-matching-related terminology used in this article 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 Figure 2, the quantified formula F_3 will be instantiated only when a *triggering term* that matches the *pattern* $\{\mathtt{Len}(s_3)\}$ 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). Patterns are matched *modulo equalities*, that is, F_4, which has the pattern $\{\mathtt{Len}(\mathtt{Build}(s_4, i_4, v_4, l_4))\}$, will be instantiated also when the solver is provided the triggering term $\mathtt{Len}(s)$ and it knows that $s = \mathtt{Build}(s_4, i_4, v_4, l_4)$ holds for some $s_4\colon \mathrm{U}, i_4\colon \mathrm{Int}, v_4\colon \mathrm{U}, l_4\colon \mathrm{Int}$. However, our algorithm does not generate such triggering terms, as it automatically substitutes s by the right-hand side of the equality.

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

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

As 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 $L = -1$. The sub-terms $\mathtt{Build}(\mathtt{Empty}(\mathtt{typ}(v))$ and $\mathtt{Len}(\mathtt{Build}(\mathtt{Empty}(\mathtt{typ}(v)), 0, v, -1))$ of the synthesized triggering term

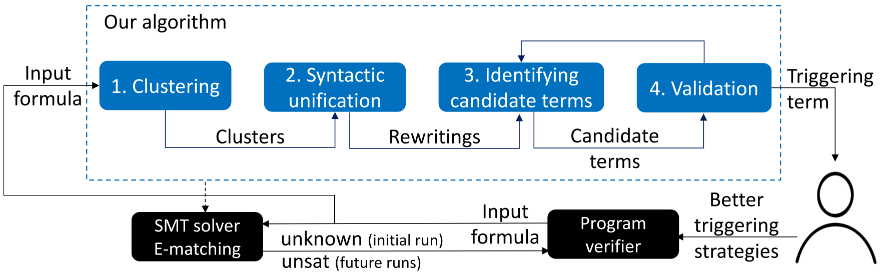

Fig. 4. Main steps of our algorithm, represented as blue boxes, which helps the developers of SMT-based verifiers devise better triggering strategies (and enable E-matching to prove unsat). The arrows depict data.

lead to the instantiation of F_2 and F_3, respectively:

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

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

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

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 $0 \le L$. Since $(L = -1) \land (0 \le L) = \texttt{false}$, the unsatisfiability proof succeeds.

3 OVERVIEW

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

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

Let F be a second-order formula, in which universal quantifiers appear only in positive positions. We define its *approximation* as

$$F_{\approx} = F[\exists\overline{\texttt{f}}\ /\ \forall\overline{\texttt{f}}], \tag{1}$$

where $\overline{\texttt{f}}$ 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 $\neg\texttt{I}_{\approx}$ and check if it is sufficient to prove that $\texttt{I}$ is unsatisfiable (due to the approximation, a model is no longer guaranteed to be a counterexample for the original formula).

The four main steps of our algorithm are depicted in Figure 4. The algorithm is stand-alone, i.e., not integrated into, nor dependent on any specific SMT solver. We illustrate it on the inconsistent

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

Fig. 5. Formulas that set contradictory constraints on the uninterpreted function f. Synthesizing the triggering term dummy(f(g(7))) requires theory reasoning and syntactic unification. dummy is a fresh uninterpreted function (see Step 4).

axioms from Figure 5 (which we assume are part of a larger axiomatization). To show that the formula $\mathtt{I} = F_0 \wedge F_1 \wedge \ldots$ 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.

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 thus identify *clusters* C of formulas F that share function symbols and then process each cluster separately. In Figure 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_i in C such that: (1) F_i is unsatisfiable by itself, or (2) F_i 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 $C_{\approx}$ is defined in (1). 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 Figure 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 issue by requiring F_0 to contradict the *instantiation* of F_1, which is a weaker constraint.

Let F be an arbitrary formula, with universal quantifiers only in positive positions. We define its *instantiation* as

$$F_{Inst} = F[\exists \overline{\mathtt{x}} \,/\, \forall \overline{\mathtt{x}}], \tag{2}$$

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

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

Fig. 6. Grammar of input formulas $\mathtt{I}$. 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.

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 `dummy`, 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 `dummy(f(g(7)))` is sufficient to complete the unsatisfiability proof.

4 SYNTHESIZING TRIGGERING TERMS

Next, we present our algorithm for synthesizing triggering terms required by E-matching to return unsat: in Section 4.1, we define the input formulas and in Section 4.2, we explain the details of the algorithm. Its extensions follow in Section 4.3. We illustrate the algorithm on additional examples in Section 4.4.

4.1 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 (Section 6 addresses scalability issues). The resulting formulas follow the grammar from Figure 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 we ensure 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.*

4.2 Algorithm

The pseudo-code of our algorithm is given in Algorithm 1. It takes as input an SMT formula `I` (defined in Figure 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 at the end of this subsection.

The algorithm iterates over each *quantified* conjunct F of `I` (Algorithm 1, line 3) and checks if it 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 Figure 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

ALGORITHM 1: Our algorithm for synthesizing triggering terms that enable unsatisfiability proofs via E-matching. We assume here that all quantified variables are globally unique and $\mathtt{I}$ does not contain nested quantifiers. We use the notation $\bigtimes S_i$ to represent the Cartesian product of the sets S_i. `checkSat` yields the solver's result on the given formula (i.e., sat, unsat, unknown, timeout, or error), a model for satisfiable formulas, and an unsat core for unsatisfiable ones (the unsat core is not relevant for our algorithm). `minimized` removes redundant (sub-)terms from a triggering term. The auxiliary procedures `clustersRewritings` and `candidateTerm` are presented in Algorithm 2 and Algorithm 3, respectively.

```
Arguments: $\mathtt{I}$ — input formula, also treated as set of conjuncts
           $\sigma$ — similarity threshold for clustering
           $\delta$ — maximum depth for clustering
           $\mu$ — maximum number of different models
Result: The synthesized triggering term or None, if no term was found

Procedure synthesizeTriggeringTerm
    foreach depth $\in \{0, \ldots, \delta\}$ do
        foreach $F \in \mathtt{I} \mid F$ is $\forall \overline{x} :: F'$ do
            foreach $(C, R) \in$ clustersRewritings($\mathtt{I}$, $F$, $\sigma$, depth) do      // steps 1, 2
                Inst $\longleftarrow \{\}$
                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
                    Inst[$f$] $\longleftarrow \{(\bigwedge_{0 \le j < k} \neg D_j) \wedge D_k \mid 0 \le k \le n\}$
                Inst[$F$] $\longleftarrow \{\neg F'\}$
                foreach $H \in \bigtimes\{$Inst[$f$] $\mid f \in \{F\} \cup C\}$ do      // $\bigtimes$ = Cartesian product
                    $G \longleftarrow \bigwedge H \wedge \bigwedge R$
                    foreach m $\in \{0, \ldots, \mu - 1\}$ do
                        resG, model $\longleftarrow$ checkSat($G$)
                        if resG $\neq$ SAT then
                            break      // no models if $G$ is not SAT
                        $T \longleftarrow$ candidateTerm($\{F\} \cup C$, $R$, model)      // step 3
                        resI, _ $\longleftarrow$ checkSat($\mathtt{I} \wedge T$)      // step 4
                        if resI = UNSAT then
                            return minimized($T$)      // success
                        $G \longleftarrow G \wedge \neg$model      // avoid this model next iteration
    return None
```

sets of uninterpreted function symbols (from their bodies and the patterns of the quantifiers). The Jaccard similarity index is defined as

$$J(F_i, F_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}. \tag{3}$$

That is, the number of common uninterpreted functions divided by the total number. For the two formulas from Figure 5, $S_0 = \{\mathtt{f}\}$, $S_1 = \{\mathtt{f}, \mathtt{g}\}$, therefore $J(F_0, F_1) = \frac{|\{\mathtt{f}\}|}{|\{\mathtt{f},\mathtt{g}\}|} = 0.5$.

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

$$\mathtt{sim}^{\delta}_{\mathtt{I}}(F_i, F_j, \sigma) = \begin{cases} J(F_i, F_j) \ge \sigma, & \delta = 1 \\ \exists F_k : \mathtt{sim}^{\delta-1}_{\mathtt{I}\setminus\{F_i\}}(F_i, F_k, \sigma) \text{ and } J(F_k, F_j) \ge \sigma, & \delta > 1 \end{cases}, \tag{4}$$

ALGORITHM 2: Auxiliary procedure for Algorithm 1, which identifies clusters of formulas similar to F and their rewritings. `sim` is defined (4). `unify` is a first-order unification algorithm (not shown); it returns a set of rewritings with restricted shape, defined in (5). `qvars` returns the set of quantified variables from the formulas of a given cluster. `lhs` represents the left-hand of a rewriting, which is, according to (5), a quantified variable.

```
Arguments: I — input formula, also treated as set of conjuncts
           $F$ — quantified conjunct of I, i.e., $F \in \mathtt{I} \mid F$ is $\forall\overline{x} :: F'$
           $\sigma$ — similarity threshold for clustering
           depth — current depth for clustering
Result: A set of pairs, consisting of clusters and their corresponding rewritings

Procedure clustersRewritings
    if depth = 0 then
        return $\{(\emptyset, \emptyset)\}$
    simFormulas ⟵ $\{f \mid f \in \mathtt{I} \setminus \{F\}$ and $\mathtt{sim}^{\text{depth}}_{\mathtt{I}}(F, f, \sigma)\}$      // step 1
    rewritings ⟵ {}
    foreach $f \in$ simFormulas do
        rws ⟵ unify$(F, f)$                                                   // step 2
        if rws = $\emptyset$ and ($f$ is $\forall\overline{x} :: D_0 \vee \ldots \vee D_n$) then
            simFormulas ⟵ simFormulas $\setminus \{f\}$
        rewritings[$f$] ⟵ rws
    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 $x = \mathtt{lhs(r)}\}| \leq 1)\}$
```

where $\sigma \in [0, 1]$ is a similarity threshold used to parameterize our algorithm and J is defined in (3).

The initial cluster (for `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 7), we identify *rewritings*, i.e., constraints under which two similar *quantified* formulas share terms. (Section 4.4 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 $\mathtt{f}$ they have the following shape:

$$x_m = rhs_n, \tag{5}$$

where $m = i$ and $n = j$ or $m = j$ and $n = i$, x_m is a quantified variable of F_m, F_m contains a term $\mathtt{f}(x_m)$, F_n contains a term $\mathtt{f}(rhs_n)$, and rhs_n is a constant c_n, a quantified variable x_n, or a composite function $(\mathtt{f} \circ \mathtt{g}_0 \circ \cdots \circ \mathtt{g}_p)(\overline{c_n}, \overline{x_n})$ occurring in the formula F_n; $\mathtt{g}_0, \ldots, \mathtt{g}_p$ are arbitrary (interpreted or uninterpreted) functions. We thus determine the *most general unifier* [14] only for those terms that have uninterpreted functions as the outer-most functions and quantified variables as arguments. The unification algorithm is standard (except for the restricted shape), so it is not shown explicitly.

In Figure 5, F_1 is similar to F_0 for any $\sigma \leq 0.5$. We then compute the rewritings for all the quantified variables of F_0 that appear in its body as arguments to some *common* uninterpreted

$$\begin{aligned} F_0: &\quad \forall x_0: \text{Int} :: \{\mathtt{f}(x_0)\}\ \mathtt{f}(x_0) = 6 \\ F_1: &\quad \forall x_1: \text{Int} :: \{\mathtt{f}(x_1)\}\ \mathtt{f}(x_1) = 7 \\ F_2: &\quad \forall x_2: \text{Int} :: \{\mathtt{f}(x_2)\}\ \mathtt{f}(x_2) = 8 \end{aligned}$$

Fig. 7. Formulas that set contradictory constraints on the uninterpreted function f. Synthesizing the triggering term dummy(f(0)) requires clusters of similar formulas with alternative rewritings.

$$\begin{aligned} F_0: &\quad \forall x_0: \text{Int} :: \{\mathtt{f}(x_0)\}\ \neg(x_0 > -1) \vee \mathtt{f}(x_0) > 7 \\ F_1: &\quad \forall x_1: \text{Int} :: \{\mathtt{f}(x_1)\}\ \neg(x_1 < 1) \vee \mathtt{f}(x_1) = 6 \end{aligned}$$

Fig. 8. Formulas that set contradictory constraints on the uninterpreted function f. Synthesizing the triggering term dummy(f(0)) requires instantiations that cover all the disjuncts.

functions (in this case, only x_0). Unifying the terms $\mathtt{f}(x_0)$ and $\mathtt{f}(\mathtt{g}(x_1))$ generates the rewriting $x_0 = \mathtt{g}(x_1)$, which has the shape defined in (5).

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 11). For example, in Figure 7, both F_1 and F_2 are similar to F_0 (all three formulas share the uninterpreted function symbol f). As 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 (1), which is equivalent to $\neg F'$, since F has the shape $\forall \overline{x} :: F'$ from Algorithm 1, line 3), (2) the *instantiations* (defined in (2)) of all the similar formulas from the cluster, and (3) the corresponding rewritings R. (As we assume that all the quantified variables are globally unique, we do not perform variable renaming when computing the instantiations).

If a similar formula has multiple disjuncts D_k, the SMT solver may use short-circuiting semantics when generating the model for G. That is, if it can find a model that satisfies the first disjunct, it may 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 last once. We thus compute *multiple instantiations* of 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 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). (For presentation purposes, we also store $\neg F'$ in the instantiations map (Algorithm 1, line 8), even if it does *not* represent the instantiation of F.)

In Figure 8, F_1 is similar to F_0 and $R = \{x_0 = x_1\}$. F_1 has two disjuncts and thus two possible instantiations: $\mathtt{Inst}[F_1] = \{x_1 \ge 1, (x_1 < 1) \wedge (\mathtt{f}(x_1) = 6)\}$. 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.

The procedure candidateTerm in Algorithm 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

ALGORITHM 3: Auxiliary procedure for Algorithm 1, which constructs a triggering term from the given cluster, rewritings, and SMT model. `patterns` and `qvars` return the set of patterns, respectively of quantified variables from the formulas of a given cluster. `dummy` is a fresh uninterpreted function symbol, which conveys no information about the truth value of the candidate term; therefore 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     $P_0, \ldots, P_k \longleftarrow$ patterns($C$)
3     while $R \neq \varnothing$ do
4         choose and remove $r \longleftarrow (x = rhs)$ from $R$
5         $P_0, \ldots, P_k \longleftarrow (P_0, \ldots, P_k)[\,rhs/x\,]$
6         $R \longleftarrow R\,[\,rhs/x\,]$
7     foreach $x \in$ qvars(C) do
8         $P_0, \ldots, P_k \longleftarrow (P_0, \ldots, P_k)[$ model(x)$/x\,]$
9     return "dummy" + "(" + $P_0, \ldots, P_k$ + ")"
```

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). For interpreted, user-defined types (such as a type `IList` for representing a `List` of `Int`, where `List` and `Int` are both interpreted types), the solver generates constants for each type component, or a sequence of operations required to construct them. For instance, `insert(0, nil)` (i.e., a singleton list containing the constant 0) is a possible model provided by the SMT solver Z3 [24] for a variable of type `IList`. 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 (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). The parameter μ allows us to deal with other sources of incompleteness, as we explain next.

Let us consider the formula from Figure 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 Z3 for the formula $G = \neg F'$ is $x = -1, y = 0$. $-1/0$ is defined ("/" is a total function [18]), but its result is not specified. Thus, the solver cannot validate this model (i.e., it returns *unknown*). In such cases, we ask the solver for a different model. In Figure 9, if we simply exclude previous models, we can

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

Fig. 9. Inconsistent axiom from F* [49]. $\texttt{\_div}\colon \text{Int} \times \text{Int} \rightarrow \text{Int}$ is an uninterpreted function. Synthesizing the triggering term $\texttt{dummy}(\texttt{\_div}(1,2))$ requires diverse models.

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; 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 Algorithm 1, line 19 as $(\bigwedge_{x \in \overline{x}} x \neq \texttt{model}(x)) \wedge (\bigwedge_{x_i, x_j \in \overline{x},\ i \neq j,\ \texttt{model}(x_i) = \texttt{model}(x_j)} x_i \neq x_j)$. This allows us to synthesize the triggering term $\texttt{dummy}(\texttt{\_div}(1,2))$ and expose the error from Figure 9.

The first component $(\bigwedge_{x \in \overline{x}} x \neq \texttt{model}(x))$ 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 (i.e., $\bigwedge_{x_i, x_j \in \overline{x},\ i \neq j,\ \texttt{model}(x_i) = \texttt{model}(x_j)} x_i \neq x_j$) 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.

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 $\texttt{f}(x_0, \texttt{skolem}())$ and $\texttt{f}(x_1, 1)$ 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.

4.3 Extensions

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

Combining multiple candidate terms. In Algorithm 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. Figure 10 presents a simple example from SMT-COMP 2019 pending benchmarks [3]. (The files in this category are not guaranteed to comply with the SMT-LIB standard, but our benchmarks selection algorithm described in Section 5.2 checks this automatically.) The input $\texttt{I} = F_0 \wedge F_1$ is unsatisfiable, as there does not exist an interpretation for the uninterpreted function $\texttt{U}$ that satisfies all the constraints: F_1 requires $\texttt{U(s)}$ to be $\texttt{true}$; if F_0 is instantiated for $x_0 = \texttt{s}$, the solver learns that $\texttt{U(il)}$ must be $\texttt{true}$ as well; however, if $x_0 = \texttt{il}$, then $\texttt{U(il)}$ must be $\texttt{false}$, which is a contradiction. Exposing the inconsistency thus requires two instantiations of F_0, triggered by $\texttt{f(s)}$ and $\texttt{f(il)}$, respectively. We generate both

$$F_0: \quad \forall x_0 : \mathsf{S} :: \{\mathtt{f}(x_0)\}\ \neg\mathsf{U}(x_0) \vee (\mathsf{U}(\mathtt{f}(x_0)) \wedge \mathtt{f}(x_0) = \mathtt{il} \wedge x_0 \neq \mathtt{il})$$
$$F_1: \quad \mathsf{U}(\mathtt{s})$$

Fig. 10. Benchmark from SMT-COMP 2019 [3]. The formulas set contradictory constraints on the uninterpreted function U. S is an uninterpreted type, s and il are user-defined constants of type S. Synthesizing the triggering term dummy(f(s), f(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.

$$F: \quad \forall x : \mathrm{Int}\ :: \{\mathtt{g}(x)\}\ \mathtt{f}(x) \neq \mathtt{f}(7)$$

Fig. 11. Formula that sets contradictory constraints on the uninterpreted function f. The uninterpreted function g is used only as a pattern (i.e., it does not appear in the body of F, see Section 4.4). Synthesizing the triggering term dummy(g(7)) requires unification across multiple instantiations.

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

Fig. 12. Fragment of Gobra's [52] option types axiomatization. U is an uninterpreted type, none is a user-defined constant of type U. F_1 and F_2 have *multi-patterns* (discussed in Section 4.4). Synthesizing the triggering term dummy(some(get(none))) requires type-based constraints.

triggering terms, but in separate iterations (independently, both fail to validate). However, by validating them simultaneously (i.e., conjoining both of them to I, as arguments to the fresh function dummy), our algorithm identifies the required triggering term T = dummy(f(s), f(il)).

Unification across multiple instantiations. The clusters constructed by our algorithm are sets (see Algorithm 2, line 11), 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 (now represented as a list) 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. Figure 11 presents an example, which consists of a single inconsistent formula F. Our regular algorithm from Algorithm 2 does not identify any rewritings. However, with this extension, F unifies with itself for any $\Phi > 1$, and one possible rewriting is $x' = 7$ (where x' is a fresh variable representing the second instantiation of F). The corresponding triggering term, T = dummy(g(7)), allows E-matching to prove unsat. Note that the uninterpreted function g is used only as a pattern. If the pattern would have been f(x), any triggering term f(c), where c is an integer constant, would have been sufficient to complete the proof: (1) for $c = 7$, the contradiction would have been exposed directly; (2) for $c \neq 7$, the term f(7), obtained from the first instantiation of F, would have triggered its second instantiation and case (1) would have then applied.

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 Figure 12, the

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

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

solver cannot provide concrete values of the uninterpreted type U for e_1 and op_1, it can only assign fresh, unconstrained variables (e.g., e and op). However, the triggering terms some(e) and get(op), which can be obtained from these fresh variables, 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 Section 4.2. 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 Figure 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 [52] option types axiomatization.

Unification for sub-terms. Figure 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*; due to our restrictive shape of the rewritings, the sub-terms can only be applications of *uninterpreted* functions. In Figure 13, trying to unify $\texttt{f}(\texttt{g}(x_0))$ does not produce any rewritings, as F_1 does not contain f(g). We thus unify the sub-term $\texttt{g}(x_0)$ with g(2020) and 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 unsat proof to succeed. This unification is syntactic, but produces the triggering terms that would be obtained if the solver would apply some uninterpreted function present in the input on a learned predicate (the solver performs semantic reasoning automatically, but without generating new triggering terms).

Alternative triggering terms. Our algorithm returns the *first* candidate term that successfully validates (Algorithm 1, line 18). However, it might also be useful to synthesize *alternative* triggering terms for the same input, as they may correspond to different completeness or soundness issues. Our tool provides this option and can also return *all* the triggering terms found within the given timeout.

All these extensions (individually or together with other extensions) allow us to complete the refutation proofs for particular benchmarks. Section 5 evaluates the impact of a few configurations of our technique, which can be obtained by enabling some of the extensions or by setting certain values for some of the additional parameters. Automatically determining which is the most suited configuration for a particular input is left as future work.

4.4 Additional Examples

In this section, we illustrate our algorithm on various examples (including those from Figure 1 and Figure 2, and an example with nested quantifiers). We also explain how the algorithm supports quantifier-free formulas, synonym functions as patterns, multi-patterns, and alternative patterns.

Nested quantifiers. Our algorithm handles inputs with nested quantifiers as described in Section 4.2. We illustrate this aspect on the formulas from Figure 14, which axiomatize operations

$$
\begin{aligned}
F_0:&\quad \forall l_0: \mathrm{L} :: \{\mathtt{isEmpty}(l_0)\}\ \neg(l_0 = \mathtt{EmptyList}) \lor \mathtt{isEmpty}(l_0)\\
F_1:&\quad \forall l_1: \mathrm{L} :: \{\mathtt{isEmpty}(l_1)\}\ \mathtt{isEmpty}(l_1) \lor \mathtt{has}(l_1, \mathtt{f}_1(l_1))\\
F_2:&\quad \forall l_2: \mathrm{L} :: \{\mathtt{isEmpty}(l_2)\}\ \neg\mathtt{isEmpty}(l_2) \lor \forall el_2: \mathrm{Int} :: \{\mathtt{has}(l_2, el_2)\}\ \neg\mathtt{has}(l_2, el_2)\\
F_3:&\quad \forall l_3: \mathrm{L}, el_3: \mathrm{Int} :: \{\mathtt{has}(l_3, el_3)\}\ \mathtt{has}(l_3, el_3) \lor (\mathtt{indexOf}(l_3, el_3) = -1)\\
F_4:&\quad \forall l_4: \mathrm{L}, el_4: \mathrm{Int} :: \{\mathtt{indexOf}(l_4, el_4)\}\ \mathtt{indexOf}(l_4, el_4) \geq 0
\end{aligned}
$$

Fig. 14. Formulas that set contradictory constraints on the uninterpreted function indexOf. L is an uninterpreted type, EmptyList is a user-defined constant of type L. $\mathtt{f}_1$ is a Skolem function, which replaces a nested existential quantifier. F_2 contains nested universal quantifiers.

$$
\begin{aligned}
F_0:&\quad \forall x_0: \mathrm{Int} :: \{\mathtt{len}(\mathtt{nxt}(x_0))\}\ \mathtt{len}(x_0) > 0\\
F_1:&\quad \forall x_1: \mathrm{Int} :: \{\mathtt{len}(\mathtt{nxt}(x_1))\}\ (\mathtt{nxt}(x_1) = x_1) \lor (\mathtt{len}(x_1) = \mathtt{len}(\mathtt{nxt}(x_1)) + 1)\\
F_2:&\quad \forall x_2: \mathrm{Int} :: \{\mathtt{len}(\mathtt{nxt}(x_2))\}\ \neg(\mathtt{nxt}(x_2) = x_2) \lor (\mathtt{len}(x_2) = 1)\\
F_3:&\quad \mathtt{len}(7) \leq 0
\end{aligned}
$$

Fig. 15. Boogie example from Figure 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 (for discharging the proof obligation, the verifier considers the axioms and the negation of the verification condition).

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 Algorithm 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: $\mathsf{Inst}[F_2] = \{\neg\mathtt{isEmpty}(l_2), \mathtt{isEmpty}(l_2) \land \neg\mathtt{has}(l_2, el_2)\}$, $\mathsf{Inst}[F_0] = \{\neg(l_0 = \mathtt{EmptyList}), l_0 = \mathtt{EmptyList} \land \mathtt{isEmpty}(l_0)\}$. The model of the corresponding G formula and R allow us to synthesize the required triggering term $T = \mathtt{dummy}(\mathtt{isEmpty}(\mathtt{EmptyList}), \mathtt{has}(\mathtt{EmptyList}, 0))$.

Quantifier-free formulas. Our algorithm 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 (Algorithm 2, line 8). Since quantifier-free conjuncts can be seen as already instantiated formulas, we only have to cover all their disjuncts (Algorithm 1, line 7).

Boogie example. Figure 15 shows the example from Figure 1 encoded in our input format. The quantifier-free formula F_3 (i.e., the negation of the verification condition) is similar to F_0 (they share the function symbol len) and unifies through the rewritings $R = \{x_0 = 7\}$. We obtained the required triggering term $T = \mathtt{dummy}(\mathtt{len}(\mathtt{nxt}(7)))$ from the model of the formula $G = \neg F_0' \land \mathsf{Inst}[F_3][0] \land \bigwedge R = (\mathtt{len}(x_0) \leq 0) \land (\mathtt{len}(7) \leq 0) \land (x_0 = 7)$.

Dafny example. Our algorithm can synthesize various triggering terms that expose the unsoundness from Figure 2, depending on the values of its parameters. We explain here one, obtained for $\sigma = 0.1$. For $\mathtt{depth} = 0$, the algorithm checks each formula F_0–F_4 in isolation. As they are all individually satisfiable, it continues with $\mathtt{depth} = 1$. To avoid redundant explanations, we present only the iteration for F_3. F_3 shares at least two uninterpreted function symbols with each of the other formulas, so there are various alternative rewritings: $s_3 = \mathtt{Empty}(t_1)$, $s_3 = s_4$, $s_3 = \mathtt{Build}(s_4, i_4, v_4, l_4)$, and so on. As we consider clusters-rewritings pairs in which each quantified variable has maximum one rewriting, one such pair is $(C = \{F_4\}, R = \{s_3 = \mathtt{Build}(s_4, i_4, v_4, l_4)\})$. F_4 has two disjuncts, therefore its instantiations are: $\mathsf{Inst}[F_4] = \{\mathtt{typ}(s_4) = \mathtt{Type}(\mathtt{typ}(v_4), \neg(\mathtt{type}(s_4) = \mathtt{Type}(\mathtt{typ}(v_4)) \land (\mathtt{Len}(\mathtt{Build}(s_4,\ i_4, v_4, l_4)) = l_4)\}$. From these instantiations and the rewritings

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

Fig. 16. Inconsistent formula from a VCC/HAVOC [22, 47] benchmark from SMT-COMP 2020 [5], which cannot be proved unsat by MBQI. Our synthesized triggering term $\texttt{dummy}(\texttt{both\_ptr}(-2, -1, 0))$ allows E-matching to refute it.

R, we derive two formulas: $G_0 = \neg F_3' \wedge \texttt{Inst}[F_4][0] \wedge \bigwedge R$, with the model $s_3 = s$, $s_4 = s'$, $i_4 = 0$, $v_4 = v$, $l_4 = 1$ and $G_1 = \neg F_3' \wedge \texttt{Inst}[F_4][1] \wedge \bigwedge R$, with the model $s_3 = s$, $s_4 = s'$, $i_4 = 0$, $v_4 = v$, $l_4 = -1$, 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 *rhs* in the rewriting, i.e., $\texttt{Build}(s_4, i_4, v_4, l_4)$, and all the other quantified variables by their constants from the model. The result after removing redundant terms is: $T_0 = \texttt{dummy}(\texttt{Len}(\texttt{Build}(t, 0, v, 1)))$ and $T_1 = \texttt{dummy}(\texttt{Len}(\texttt{Build}(t, 0, v, -1)))$. 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 expands 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 $(C = \{F_4\}, R = \{s_3 = \texttt{Build}(s_4, i_4, v_4, l_4)\})$ can be extended to $(C = \{F_4, F_1\}, R = \{s_3 = \texttt{Build}(s_4, i_4, v_4, l_4), s_4 = \texttt{Empty}(t_1), t_1 = \texttt{typ}(v_4)\})$, as F_1 is similar to 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: $s_3 = s$, $s_4 = s'$, $i_4 = 0$, $v_4 = v$, $l_4 = -1$, $t_1 = t$, where s, s', v, and t are fresh variables of types U and V, respectively. From this model and the rewritings, we construct the triggering term $T = \texttt{dummy}(\texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)))$, which is sufficient to expose the inconsistency between F_3 and F_4.

VCC/HAVOC example. Figure 16 presents a fragment of an unsatisfiable benchmark generated by VCC/HAVOC [22, 47], which cannot be refuted by neither E-matching, nor MBQI. (Section 5 provides additional experimental results and a detailed comparison between our algorithm and alternative refutation techniques.) F, which was part of a set of 160 formulas, is inconsistent by itself: when $size = 0$, it evaluates to false for any integer values a, b, such that $a \leq b$. Our algorithm synthesizes a triggering term for E-matching in $\approx$7 s 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 $size = 0$ and some values for a and b, such that $a \leq b$ (e.g., $a = -2$, $b = -1$).

Synonym functions as patterns. For the examples discussed so far, the functions used as patterns were also present in the body of the quantifiers. However, to have 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 patterns are not uncommon in proof obligations. Figure 17 shows an example, which uses the synonym functions technique [33] to avoid matching loops. sum and sum_syn compute the sum of the elements of a sequence, between a 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

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

Fig. 17. Formulas with synonym functions as patterns that axiomatize sequence comprehensions and set contradictory constraints on the uninterpreted function sum_syn. ISeq is a user-defined type, empty is a user-defined constant of type ISeq (i.e., the empty sequence).

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

Fig. 18. Formulas that set contradictory constraints on the uninterpreted function f. B is an uninterpreted type. F_1 has a multi-pattern.

returns the nth 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 $l_0 = l_2 + 1$. The 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 Figure 18 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 Algorithm 3, line 9). For the example from Figure 18, 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 $G = \neg F_0' \wedge \mathsf{Inst}[F_1][1] \wedge \bigwedge R = (\texttt{f}(x_0) = 7) \wedge (\neg\texttt{g}(b_1) \wedge \texttt{f}(x_1) = x_1) \wedge (x_0 = x_1)$. Here b is a fresh, unconstrained variable of the uninterpreted type B.

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

5 EVALUATION

Evaluating our work requires benchmarks with known triggering issues (i.e., for which E-matching yields *unknown* due to incomplete quantifier instantiations). Since there is no publicly available suite, in Section 5.1, we used manually-collected benchmarks from four verifiers [32, 38, 49, 52]. Our algorithm succeeded for 65.6%. To evaluate its applicability to other verifiers, in Section 5.2, we used SMT-COMP [5] inputs. As they were not designed to expose triggering issues, we developed a filtering step to automatically identify the subset that falls into this category. The results show that our algorithm is suited also for benchmarks from Spec# [16], Havoc [22], and VCC [47]. Section 5.3 illustrates that our triggering terms are simpler than the unsat proofs produced

Table 1. Results on Verification Benchmarks with Known Triggering Issues. The columns from left to right show: the source of the benchmarks, the number of files (#), their number of conjuncts (#F) and of quantifiers (#∀), the number of files for which five different configurations of our algorithm (**C0–C4**) synthesized suited triggering terms, our results across all configurations, the number of unsat proofs generated by Z3 (with MBQI [28]), CVC4 (with enumerative instantiation [43]), and Vampire [31] (in CASC mode [48], using Z3 for ground theory reasoning). The columns marked with grey represent E-matching-based algorithms; only those can be soundly used by verifiers whose SMT encodings have patterns, i.e., are designed for E-matching

	#	#F	#∀	C0	C1	C2	C3	C4	Our	Z3	CVC4	Vampire
Source		min-max	min-max	default	$\sigma = 0.1$	$\beta = 1$	type	$\sigma = 0.1 \wedge$ sub	work	MBQI	enum inst	CASC ∧ Z3
Dafny	4	6–16	5–16	1	1	1	1	0	**1**	1	0	2
F*	2	18–2,388	15–2,543	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	**32**								**21 (65.6%)**	**19 (59.3%)**	**0 (0%)**	**30 (93.7%)**

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

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 open-source tool [7] 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 the following values: `sat.random_seed` to 488, `smt.random_seed` to 599, `nlsat.seed` to 611. We set the (soft) timeout to 600 s and the memory limit to 6 GB per run and used a 1 s 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 and can be replicated via our Docker image [6].

5.1 Effectiveness on Verification Benchmarks with Known Triggering Issues

First, we used *manually*-collected benchmarks with known triggering issues from four state-of-the-art verifiers: Dafny [32], F* [49], Gobra [52], and Viper [38]. 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 [1], with at least one assertion needed only for triggering (we removed these assertions from the benchmarks, as our work is expected to find the triggering terms automatically). The Viper files contain algorithms for arrays, binomial heaps, binary search trees, and regression tests. The input sizes (minimum–maximum number of formulas or quantifiers) are shown in Table 1, columns #F–#∀.

Configurations. We ran our tool with five configurations, to also analyze the impact of its parameters (see Algorithm 1 and Section 4.3). 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 Table 1. All configurations use $\delta = 2$ (maximum transitivity depth), $\mu = 4$ (maximum number of different models), and 600 s timeout per file.

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

Source	#	#*F* min-max	#∀ min-max	C0 default	C1 $\sigma = 0.1$	C2 $\beta = 1$	C3 type	C4 $\sigma = 0.1 \wedge$ sub	Our work	Z3 MBQI	CVC4 enum inst	Vampire CASC ∧ Z3
Spec#	33	28–2,363	25–645	16	16	14	16	15	**16**	16	0	29
VCC/Havoc	14	129–1,126	100–1,027	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. Columns **C0**–**C4** in Table 1 show the number of files solved by each configuration, **Our work** summarizes those 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 ≈60 s is given in Figure 9. It required two developers to be manually diagnosed based on a bug report [4]. A simplified Gobra axiomatization for option types is shown in Figure 12; the entire axiomatization (considered in Table 1) was solved only by C4 in ≈13 s. 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, we believe 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 [2], 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` (`val` allows one to access the value at the corresponding heap location). Both fixes result in E-matching refuting the proof obligation in under 0.1 s. We also exposed another case where Boogie (which is used by Viper) is sound only modulo patterns (as in Figure 3), i.e., the unsoundness is visible only at the SMT level.

As Table 1 shows, configurations with smaller σ (C1 and C4) were particularly important for some of the F* and Gobra benchmarks. Our algorithm starts with the given σ and if it does not find the required triggering terms, it decreases σ by 0.1 and reiterates. Thus C0 also covers the case $\sigma = 0.1$, if the overall timeout is large enough. However, always starting with a small σ may prevent our algorithm from synthesizing the triggering terms, since the number of rewritings it has to explore is considerably high. The extensions for unifying sub-terms (C4) and identifying type-based constraints (C3) were also needed for one, respectively, two input files.

5.2 Effectiveness on SMT-COMP Benchmarks

Next, we considered 61 SMT-COMP [5] benchmarks from Spec# [16], VCC [47], Havoc [22], Simplify [25], and the Bit-Width-Independent (BWI) encoding [39]. These were selected *automatically* using a filtering algorithm that we designed (described below) and are summarized in Table 2.

Benchmarks selection. We collected *all* 27,716 benchmarks from SMT-COMP 2020 (single query track) [5], 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 600 s per file and 4 s per formula. We also removed the files with features not yet supported by PySMT [27], 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 61 examples that could not be solved within 600 s due to incompleteness in instantiating quantifiers (our work only targets this incompleteness, but the SMT-COMP suite also contains other solving challenges).

Results. The results are shown in Table 2, which follows the structure of Table 1. 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 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. The other configurations did not prove to be better than C0 for these SMT-COMP inputs.

5.3 Comparison with Unsatisfiability Proofs

As an alternative to our work, the developers of program verifiers could try to *manually* identify triggering issues from refutation proofs. In this experiment, we considered three state-of-the-art provers that rely on different solving strategies and could generate such proofs: Z3 (4.8.10, the same version as for our tool) with MBQI [28] (a model based quantifier instantiation technique), CVC4 (1.6) [17] with enumerative instantiation [43] (an algorithm based on E-matching, used when E-matching saturates), and the first-order theorem prover Vampire (4.4) [31], using Z3 for ground theory reasoning [42] and the CASC [48] portfolio mode with competition presets. Note that the only approach comparable with ours is enumerative instantiation; MBQI and Vampire *do not* consider patterns, thus they solve a *different* problem. We are not aware of any other work that synthesizes triggering terms for E-matching.

The last three columns in Tables 1 and 2 show the number of unsatisfiability proofs produced by each of these alternatives. CVC4 failed for all examples (it cannot construct proofs for quantified logics), Vampire refuted most of them. Our algorithm enabled E-matching to solve more inputs than MBQI for F* and Gobra and had similar results for Dafny, Spec#, and VCC/Havoc. All our five configurations solved two VCC/Havoc files not refuted by MBQI (Figure 16 presents one).

In terms of complexity, our triggering terms are much simpler than the proofs and directly highlight the root cause of the issues. The term `loc(a,2)` generated for Viper's array maximum example from Section 5.1 is easier to understand than MBQI's proof (which has 2,135 lines and over 700 reasoning steps) and than Vampire's proof (with 348 lines and 101 inference steps). Other proofs have similar sizes. Therefore, determining the source of the inconsistency from such proofs requires expert knowledge of the tool-specific proof format and significant manual effort.

Most deductive verifiers [10, 13, 16, 22, 26, 32, 50] employ E-matching for discharging their proof obligations because E-matching is the most efficient SMT algorithm for program verification [28] (the vast majority of the SMT-COMP benchmarks we initially collected were also directly refuted by E-matching). It is thus 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 Figure 3).

As our algorithm accepts as input an SMT formula, it can also produce triggering terms required only at the SMT level, but which cannot be encoded into the input language of the verifier (e.g., Boogie), since they are rejected by the type system. However, such triggering terms can be filtered out, as lifting them to the input language is mostly straightforward (we performed this step manually in our experiments, to identify the cases of soundness modulo patterns; automating this process is a possible future extension). However, this is not the case for refutation proofs, whose back translation to the source language is an open research problem. To enable the developers debug the axiomatizations or fix the incompleteness more efficiently, our tool can also generate *multiple* triggering terms (as explained in Section 4.3). It can thus reveal *multiple* triggering issues for the same input formula, information which cannot be directly obtained from unsatisfiability proofs.

5.4 Threats to Validity

We identified the following two threats to the validity of our experiments:

Non-determinism. The SMT solvers use randomized algorithms, which can cause non-determinism. To mitigate this problem, we fixed all the available random seeds and used the same seeds in all the phases of our evaluation (i.e., for inferring the patterns, pre-filtering via E-matching, running our tool and MBQI).

Benchmarks selection. We relied on Z3's E-matching algorithm to select examples with incompleteness in instantiating quantifiers. An implementation of E-matching from another solver could have led to different files. To avoid biases, we used Z3 in all the experiments.

6 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 Figure 6 allows us to simplify the presentation of the algorithm. However, eliminating conjunctions by applying distributivity and splitting (as described in Section 4.1) 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` tactic (used in our evaluation from Section 5), 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 ensure that all the resulting G formulas are still quantifier-free. The number of conjuncts and the number of quantifiers reported in Tables 1 and 2 were computed *before* applying distributivity, thus they are not artificially increased.

Rewritings. The restrictive shape of our rewritings (see (5)), ensures 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 $\{x_i \rightarrow x_j, x_j \rightarrow x_i\}$ [14]. (Such substitutions are rewritings of shape (5), 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 instantiations, 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 (Algorithm 1, line 9) since it can also have a high number of elements. However, many of them are in practice unsatisfiable; 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 ¬`model` from Algorithm 1, line 19) to ensure the solver does not provide previously-seen models for the quantified variables from the same set of patterns.

7 LIMITATIONS

In the following, we discuss the limitations of our approach, 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 explained in Section 3).

SMT solvers. Our algorithm synthesizes triggering terms as long as the solver can find models for our quantifier-free formulas. However, solvers are incomplete, i.e., they can return *unknown* and produce only *partial models*, which are not guaranteed to be correct. Nonetheless, we use also partial models, as the validation step (step 4 in Figure 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 $\forall x\colon \text{Int}, y\colon \text{Int} :: x = y$ 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 [15]).

Intended behavior. Our technique can detect soundness errors in axiomatizations, but it cannot check if the given axioms correctly model the *intended* behavior of the uninterpreted functions. For instance, the formula $F = \forall t\colon \text{V} :: \{\mathtt{Empty}(t)\}\ \mathtt{Len}(\mathtt{Empty}(t)) = 7$ is satisfiable, as the solver can find an interpretation for the uninterpreted functions `Empty` and `Len` (representing empty sequences and the length of a sequence, respectively). Our algorithm is thus not applicable. Nonetheless, F wrongly axiomatizes empty sequences, whose length should be 0, for all the possible types. Approaches that use non-axiomatic semantics, such as VST [12] or CompCert [36], could address this problem, which is orthogonal to our work.

Despite these limitations, our algorithm effectively identifies the triggering terms required in practical examples, as we have experimentally shown in Section 5.

8 RELATED WORK

To the best of our knowledge, no other approach automatically produces the information needed by the developers of program verifiers 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 (as we have shown in Section 5.3).

Quantifier instantiation techniques. ***Model-based quantifier instantiation*** **(MBQI)** [28] was designed for satisfiable 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* [44] is a technique for satisfiable 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 the soundness of the axiomatization.

E-matching-based approaches. Rümmer [46] 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* [43] 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 Section 5.3). 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; we use also syntactic information (obtained from the rewritings) to construct complex triggering terms.

Theorem provers. First-order theorem provers (e.g., Vampire [31]) also generate refutation proofs. More recent works combine a superposition calculus with theory reasoning [42, 51], 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.

Detecting matching loops. Becker et al. [19] dynamically detect performance issues in quantified SMT formulas that already include triggering terms, by identifying too *permissive* patterns that lead to matching loops. Our work targets soundness and completeness errors and synthesizes the triggering terms required to refute SMT inputs with overly *restrictive* patterns.

Testing verifiers. As formally verifying state-of-the-art program analysis tools (i.e., static analyzers, program verifiers) is rarely possible in practice [11, 21], a few research efforts focus on testing them. Ahn and Denney [9] proposed an approach that identifies inconsistencies in the quantified axiomatizations *without* patterns of a verifier. The work also requires a computational model of the axioms, which includes interpretations for all the function symbols. Thus, it cannot be applied to axiomatizations with uninterpreted functions and types, which are very common in program verification. The technique tests each axiom in isolation, so it cannot find non-trivial inconsistencies caused by the interaction between axioms. Our approach is fully automatic and detects complex errors by identifying sharing constraints between formulas and synthesizing triggering terms from clusters of similar formulas. Recent work by Irfan et al. [30] tests the soundness and precision of the Dafny verifier [32] by generating annotated random programs, which, by construction, fulfill or violate their specifications. The approach detects an error if the verifier accepts a program it should reject or vice versa; this decision is based on an SMT solver being able to refute the corresponding SMT formula or to find a model for it. However, the technique does not address the case in which the solver returns *unknown*. Our work provides a solution for incomplete quantifier instantiations.

9 CONCLUSIONS

In this article, we presented the first automated technique that enables the users and the developers of verifiers remedy the effects of overly restrictive patterns. Since discharging proof obligations and identifying inconsistencies in axiomatizations require the 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. This article focuses on the applications of our algorithm to program verification. However, our technique is suited also for other kinds of first-order constraint-solving problems, e.g., system-level modeling via Event-B [8] and the Isabelle/HOL [40] backend Sledgehammer [41]. (A thorough evaluation of our tool on systems beyond program verifiers is left as future work.) To tackle such problems, solvers generally require additional guidance (typically, in the form of syntactic patterns) to decide how the space of possible quantifier instantiations can be pruned, resulting in tractable, yet incomplete, search strategies. This article offers a systematic approach to gradually improve their completeness. As future work, we also plan to extend the syntactic unification, to efficiently support commonly-used interpreted functions and to avoid generating triggering terms that cause matching loops. Automatically determining the best combination of parameters (i.e., the best configuration of our algorithm) for a specific input formula is another research direction we would like to explore in the future. We also plan to investigate if our triggering terms could be used to identify potential fixes for unsound axiomatizations or to guide the developers in devising new, sound ones.

ACKNOWLEDGMENTS

We would like to thank the anonymous FAC’22 journal reviewers, as well as the FM’21 reviewers for their insightful comments and suggestions. We are also grateful to Felix Wolf for providing us the Gobra benchmarks, and to Evgenii Kotelnikov for his detailed explanations about Vampire.

REFERENCES

[1] 2013. *Viper Test Suite*. Retrieved from https://github.com/viperproject/silver/tree/master/src/test/resources. Accessed on May 4, 2021.

[2] 2015. *Array Maximum, by Elimination*. Retrieved from http://viper.ethz.ch/examples/max-array-elimination.html. Accessed on May 6, 2021.

[3] 2019. *The 14th International Satisfiability Modulo Theories Competition (Including Pending Benchmarks)*. Retrieved from https://smt-comp.github.io/2019/, https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks-tmp/benchmarks-pending. Accessed on May 14, 2020.

[4] 2019. *F* Issue 1848*. Retrieved from https://github.com/FStarLang/FStar/issues/1848. Accessed on May 6, 2021.

[5] 2020. *The 15th International Satisfiability Modulo Theories Competition*. Retrieved from https://smt-comp.github.io/2020/. Accessed on May 6, 2021.

[6] 2022. *Our Docker Image*. Retrieved from https://hub.docker.com/r/aterga/smt-triggen. Accessed on August 21, 2022.

[7] 2022. *Our Tool*. Retrieved from https://github.com/alebugariu/smt-triggen. Accessed on August 21, 2022.

[8] Jean-Raymond Abrial. 2010. *Modeling in Event-B: System and Software Engineering* (1st ed.). Cambridge University Press.

[9] Ki Yung Ahn and Ewen Denney. 2010. Testing first-order logic axioms in program verification. In *Tests and Proofs*. Gordon Fraser and Angelo Gargantini (Eds.), Springer, Berlin, 22–37.

[10] Afshin Amighi, Stefan Blom, and Marieke Huisman. 2016. VerCors: A layered approach to practical verification of concurrent software. In *Proceedings of the 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE Computer Society, 495–503. Retrieved from https://ieeexplore.ieee.org/abstract/document/7445381.

[11] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP’17)*. Association for Computing Machinery, New York, NY, 31–36. DOI: https://doi.org/10.1145/3088515.3088521

[12] Andrew W. Appel. 2011. Verified software toolchain. In *Programming Languages and Systems*. Gilles Barthe (Ed.), Springer, Berlin, 1–17.

[13] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification, In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’19). *Proceedings of the ACM on Programming Languages* 3, 147:1–147:30. DOI: https://doi.org/10.1145/3360573

[14] Franz Baader and Wayne Snyder. 2001. Unification theory. In *Handbook of Automated Reasoning*. John Alan Robinson and Andrei Voronkov (Eds.), Elsevier and MIT Press, 445–532.

[15] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO'05) (Lecture Notes in Computer Science, Vol. 5)*. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Springer, 364–387.

[16] Michael Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and verification: The Spec# experience. *Communications of the ACM* 54, 6 (June 2011), 81–91.

[17] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*. Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Springer, Berlin, 171–177.

[18] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[19] Nils Becker, Peter Müller, and Alexander J. Summers. 2019. The axiom profiler: Understanding and debugging SMT quantifier instantiations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19) (LNCS, Vol. 11427)*. Tomás Vojnar and Lijun Zhang (Eds.), Springer-Verlag, 99–116.

[20] Alexandra Bugariu, Arshavir Ter-Gabrielyan, and Peter Müller. 2021. Identifying overly restrictive matching patterns in SMT-based program verifiers. In *Formal Methods*. Marieke Huisman, Corina Păsăreanu, and Naijun Zhan (Eds.), Springer International Publishing, Cham, 273–291.

[21] Cristian Cadar and Alastair F. Donaldson. 2016. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE'16)*. Association for Computing Machinery, New York, NY, 765–768. DOI:https://doi.org/10.1145/2889160.2889206

[22] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. 2007. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems*. Orna Grumberg and Michael Huth (Eds.), Springer, Berlin, 19–33.

[23] Ádám Darvas and K. Rustan M. Leino. 2007. Practical reasoning about invocations and implementations of pure methods. In *Fundamental Approaches to Software Engineering (FASE'07) (LNCS, Vol. 4422)*. Matthew B. Dwyer and Antónia Lopes (Eds.), Springer-Verlag, 336–351.

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. C. R. Ramakrishnan and Jakob Rehof (Eds.), Springer, Berlin, 337–340.

[25] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A theorem prover for program checking. *Journal of the ACM* 52, 3 (May 2005), 365–473. DOI:https://doi.org/10.1145/1066100.1066102

[26] Marco Eilers and Peter Müller. 2018. Nagini: A static verifier for python. In *Computer Aided Verification (CAV'18) (LNCS, Vol. 10982)*. Hana Chockler and Georg Weissenbacher (Eds.), Springer International Publishing, 596–603. DOI:https://doi.org/10.1007/978-3-319-96145-3_33

[27] Marco Gario and Andrea Micheli. 2015. PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In *Proceedings of the SMT Workshop 2015*.

[28] Yeting Ge and Leonardo de Moura. 2009. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification*. Ahmed Bouajjani and Oded Maler (Eds.), Springer, Berlin, 306–320.

[29] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. 2013. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *European Conference on Object-Oriented Programming (ECOOP'13) (Lecture Notes in Computer Science, Vol. 7920)*. Giuseppe Castagna (Ed.), Springer, 451–476.

[30] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. 2022. Testing dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA'22)*. Association for Computing Machinery, New York, NY, 556–567. DOI:https://doi.org/10.1145/3533767.3534382

[31] Laura Kovács and Andrei Voronkov. 2013. First-order theorem proving and vampire. In *Computer Aided Verification*. Natasha Sharygina and Helmut Veith (Eds.), Springer, Berlin, 1–35.

[32] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Edmund M. Clarke and Andrei Voronkov (Eds.), Springer, Berlin, 348–370.

[33] K. Rustan M. Leino and Rosemary Monahan. 2009. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC'09)*. Association for Computing Machinery, New York, NY, 615–622. DOI:https://doi.org/10.1145/1529282.1529411

[34] K. Rustan M. Leino and Peter Müller. 2008. Verification of equivalent-results methods. In *European Symposium on Programming (ESOP'08) (Lecture Notes in Computer Science, Vol. 4960)*. S. Drossopoulou (Ed.), Springer-Verlag, 307–321.

[35] K. Rustan M. Leino and Philipp Rümmer. 2010. A polymorphic intermediate verification language: Design and logical encoding. In *Tools and Algorithms for the Construction and Analysis of Systems*. Javier Esparza and Rupak Majumdar (Eds.), Springer, Berlin, 312–327.

[36] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM* 52, 7 (Jul 2009), 107–115. DOI: https://doi.org/10.1145/1538788.1538814

[37] Michał Moskal. 2009. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. ACM, 20–29.

[38] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'16) (LNCS, Vol. 9583)*. B. Jobstmann and K. R. M. Leino (Eds.), Springer-Verlag, 41–62.

[39] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark Barrett, and Cesare Tinelli. 2019. Towards bit-width-independent proofs in SMT solvers. In *Automated Deduction—CADE 27*. Pascal Fontaine (Ed.), Springer International Publishing, Cham, 366–384.

[40] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer.

[41] Lawrence C. Paulson and Kong Woei Susanto. 2007. Source-level proof reconstruction for interactive theorem proving. In *Theorem Proving in Higher Order Logics*. Klaus Schneider and Jens Brandt (Eds.), Springer, Berlin, 232–245.

[42] Giles Reger, Nikolaj Bjorner, Martin Suda, and Andrei Voronkov. 2016. AVATAR modulo theories. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence (EPiC Series in Computing, Vol. 41)*. Christoph Benzmüller, Geoff Sutcliffe, and Raul Rojas (Eds.), EasyChair, 39–52. DOI: https://doi.org/10.29007/k6tp

[43] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. 2018. Revisiting enumerative instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems*. Dirk Beyer and Marieke Huisman (Eds.), Springer International Publishing, Cham, 112–131.

[44] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Computer Aided Verification*. Daniel Kroening and Corina S. Pasareanu (Eds.), Springer International Publishing, Cham, 198–216.

[45] Arsenii Rudich, Ádám Darvas, and Peter Müller. 2008. Checking well-formedness of pure-method specifications. In *Formal Methods (FM) (Lecture Notes in Computer Science, Vol. 5014)*. J. Cuellar and T. Maibaum (Eds.), Springer-Verlag, 68–83.

[46] Philipp Rümmer. 2012. E-Matching with free variables. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Nikolaj Bjørner and Andrei Voronkov (Eds.), Springer, Berlin, 359–374.

[47] Wolfram Schulte. 2008. VCC: Contract-based modular verification of concurrent C. In *Proceedings of the 31st International Conference on Software Engineering, ICSE'09* (31st international conference on software engineering, icse 2009 ed.). IEEE Computer Society. Retrieved from https://www.microsoft.com/en-us/research/publication/vcc-contract-based-modular-verification-of-concurrent-c/.

[48] Geoff Sutcliffe. 2016. The CADE ATP system competition—CASC. *AI Magazine* 37, 2 (2016), 99–101.

[49] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. Association for Computing Machinery, New York, NY, 256–270. DOI: https://doi.org/10.1145/2837614.2837655

[50] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. 387–398. Retrieved from https://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/.

[51] Andrei Voronkov. 2014. AVATAR: The architecture for first-order theorem provers. In *Computer Aided Verification*. Armin Biere and Roderick Bloem (Eds.), Springer International Publishing, Cham, 696–710.

[52] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, Joao C. Pereira, and Peter Müller. 2021. Gobra: Modular specification and verification of go programs. In *Computer Aided Verification (CAV'21)*. Alexandra Silva and R. Leino (Eds.), Springer International Publishing, 367–379.

Received 31 March 2022; revised 20 August 2022; accepted 13 November 2022