The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations

Nils Becker, Peter Müller, and Alexander J. Summers(✉)

Department of Computer Science, ETH Zurich, Zurich, Switzerland
nbecker@student.ethz.ch, {peter.mueller,alexander.summers}@inf.ethz.ch

Abstract. SMT solvers typically reason about universal quantifiers via E-matching: syntactic matching patterns for each quantifier prescribe shapes of ground terms whose presence in the SMT run will trigger quantifier instantiations. The effectiveness and performance of the SMT solver depend crucially on well-chosen patterns. Overly restrictive patterns cause relevant quantifier instantiations to be missed, while overly permissive patterns can cause performance degradation including non-termination if the solver gets stuck in a matching loop. Understanding and debugging such instantiation problems is an overwhelming task, due to the typically large number of quantifier instantiations and their non-trivial interactions with each other and other solver aspects. In this paper, we present the Axiom Profiler, a tool that enables users to analyse instantiation problems effectively, by filtering and visualising rich logging information from SMT runs. Our tool implements novel techniques for automatically detecting matching loops and explaining why they repeat indefinitely. We evaluated the tool on the full test suites of five existing program verifiers, where it discovered and explained multiple previously-unknown matching loops.

1 Introduction

SMT solvers are in prevalent use for a wide variety of applications, including constraint solving, program synthesis, software model checking, test generation and program verification. They combine highly-efficient propositional reasoning with natively supported theories and first-order quantifiers. Quantifiers are used frequently, for instance, to model additional mathematical theories and other domain-specific aspects of an encoded problem. In a program verification setting, for example, one might model a factorial function using an uninterpreted function `fact` from integers to integers and (partially) defining its meaning by means of quantified formulas such as $\forall$`i:Int :: i > 1 ⇒ fact(i) = i * fact(i-1)`.

The support for quantifiers in SMT is not without a price; satisfiability of SMT assertions with quantifiers is undecidable in general. SMT solvers employ a range of heuristics for quantifier instantiation, the most widely-used (and the one focused on in this paper) being *E-matching* [7]. The E-matching approach

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 99–116, 2019.
https://doi.org/10.1007/978-3-030-17462-0_6

attaches syntactic *patterns* to each universal quantifier, prescribing shapes of ground terms which, when encountered during the SMT solver's run, will trigger[1] a quantifier instantiation. For example, the pattern `{fact(i)}` on the quantifier above would indicate that a quantifier instantiation should be made whenever a function application `fact(t)` (for some term `t`) is encountered; the term `t` then prescribes the corresponding instantiation for the quantified variable.

The success of E-matching as a quantifier instantiation strategy depends crucially on well-chosen patterns: poorly chosen patterns can result in *too few* quantifier instantiations and failure to prove unsatisfiability of a formula, or *too many* quantifier instantiations, leading to poor and unpredictable performance, and even non-termination. For the factorial example above, a ground term `fact(n)` will match the pattern `{fact(i)}`, yielding an instantiation of the quantifier body, which includes the ground term `fact(n-1)`; this term again matches the pattern, and, if it is never *provable* that `n - x > 1` is definitely false, this process continues generating terms and quantifier instantiations indefinitely, in a *matching loop*.

Choosing suitable matching patterns is one of the main difficulties in using E-matching effectively [14,16]. It is extremely difficult to analyse *how* and *why* quantifier instantiations misbehave, especially for SMT problems with a large number of quantifiers[2]. Some solvers report high-level statistics (e.g. total number of quantifier instantiations); these are insufficient to determine whether quantifiers were instantiated as intended, and what the root causes of unintended instantiations are. SMT problems with poor performance are typically highly *brittle* with respect to changes in the input (due to internal pseudo-random heuristics), making performance problems difficult to reproduce or minimise; altering the example often unpredictably changes its behaviour. Conversely, problems with poor quantifier instantiation behaviour are not *always* slow; slowdowns typically manifest only when sufficiently many *interactions* with other aspects of the solver (e.g. theory reasoning) arise: extending problematic examples can cause sudden performance degradation, while the underlying cause existed in the original problem. There is therefore a clear need for tool support for uncovering, understanding and debugging quantifier instantiations made during SMT queries.

In this paper, we present the *Axiom Profiler*, a tool that addresses these challenges, providing comprehensive support for the manual and automated analysis of the quantifier instantiations performed by an SMT solver run, enabling a user to uncover and explain the underlying causes for quantifier-related problems. Our tool takes a log file generated by an SMT solver (in our case, Z3 [6]), interprets it, and provides a wide array of features and algorithms for displaying, navigating and analysing the data. Specifically, we present the following key contributions:

[1] In some tools, patterns are themselves alternatively called *triggers*.

[2] Such problems are common in e.g. program verification: for example, queries generated from Dafny's [13] test suite include an average of 2,500 quantifiers; it is not uncommon for hundreds to be instantiated hundreds of times for a single query: *cf.* Sect. 7.

1. We propose a *debugging recipe*, identifying the essential information needed and typical steps performed to analyse quantifier-related problems (Sect. 3).
2. We devise and present detailed *justifications* for each quantifier instantiation, including equality reasoning steps that enable the pattern match (Sect. 4).
3. We define an *instantiation graph* which reflects the causal relationships between quantifier instantiations, that is, which instantiations generate terms or equalities used to trigger which other instantiations (Sect. 5).
4. We present a novel automatic analysis over the causal graph which detects matching loops and explains why they occur (Sect. 6).
5. We provide an implementation. Our evaluation on test suites from five existing program verifiers reveals and explains (confirmed) previously-unknown matching loops (Sect. 7).

Our implementation extends the *VCC Axiom Profiler* [16], developed during the VCC [3] project at Microsoft Research. While this older tool (as well as the prior logging mechanism implemented for Z3) has been invaluable as a basis for our implementation, the features and contributions presented in this paper did not exist in the prior tool (aside from a basic explanation of single instantiations, omitting, e.g., equality reasoning steps used to justify a match). Our tool is open source and is available at https://bitbucket.org/viperproject/axiom-profiler/.

2 Background and Running Example

SMT Solving. SMT solvers handle input problems expressed as first-order logic assertions, including both *uninterpreted* function symbols and combinations of natively-supported *interpreted theories* (e.g., integers or reals). SMT problems can contain free uninterpreted symbols (e.g., unknown constants); the problem of SMT solving is to decide whether some interpretation for these symbols results in a model of the assertion (the assertion is *satisfiable*), or not (it is *unsatisfiable*).

The core of an SMT solver is a boolean SAT solving engine, which searches for a model by case-splitting on boolean literals, building up a *candidate model*. This core engine natively represents only quantifier-free propositional logic (including uninterpreted function symbols and equality). Transitive reasoning about equalities, as well as congruence closure properties (i.e., $a = b \Rightarrow f(a) = f(b)$ for functions f), is handled using an *E-graph* data structure [7], which efficiently represents the equivalence classes induced by currently-assumed equality facts (and represents disequality facts) over terms in the candidate model.

Running Example. Figure 1 shows our running example, an SMT query including a simplified modelling of program heaps and arrays, along with assertions (facts) encoding several properties: injectivity of the `slot` mapping (from integers to array locations), meaning of a `next` function (C-style pointer increment), and sortedness of an array `a`. The last two assertions represent an index `i` being somewhere early in array `a`, and an attempt to *prove* that the next array entry cannot be smaller than that at `i`; this proof goal is negated: any model found by the SMT solver is a counterexample to the proof goal. The `check-sat` command tells the solver to try to find a model for the conjunction of the assertions.

```
1  ; ... uninterpreted sorts: Heap, Loc, Arr
2  (declare-fun slot (Arr Int) Loc)          ; heap location for array slot
3  (declare-fun lookup (Heap Loc) Int)       ; dereference on the heap
4  (declare-fun next (Loc) Loc)              ; next slot: pointer increment
5  (assert ∀ar:Arr, i: Int, k:Int :: {slot(ar,i),slot(ar,k)}
6   i = k ∨ slot(ar,i) != slot(ar,k))        ; injectivity of slot (Q_inj)
7  (assert ∀ar:Arr, i: Int :: {slot(ar,i)}
8   next(slot(ar,i)) = slot(ar,i+1))         ; definition of next (Q_nxt)
9  ; ... declare uninterpreted constants h : Heap, a : Arr, len,j : Int
10 (assert ∀i: Int. {lookup(h, slot(a,i))}  ; sortedness property (Q_srt)
11  i < 0 ∨ i >= len ∨ lookup(h, slot(a,i)) >= lookup(h, next(slot(a,i))))
12 (assert 0 <= j ∧ j+100 < len)  ; avoids trivial models (e.g., len = 0)
13 (assert (not (lookup(h, slot(a,j)) > (lookup(h, next(slot(a,j))))))))
14 (check-sat)
```

Fig. 1. Running example: a simple SMT encoding of a problem with heaps and arrays. We use pseudocode roughly based on the smtlib format, with presentational liberties.

Quantifier Instantiation via E-matching. The most commonly-employed method for supporting first-order quantifiers in an SMT solver is *E-matching* [7]. Each ∀-quantified subformula (after conversion to negation normal form) must be equipped with at least one *pattern*: a set of terms to pattern-match against in order to trigger a quantifier instantiation. We write patterns in braces preceding a quantifier body, such as `{slot(ar,i)}` in line 7 of Fig. 1. This pattern prescribes instantiating the quantifier when a ground term of the form `slot(ar′,i′)` is present (for some terms `ar′`, `i′`) in the E-graph. In this instantiation, `ar` is bound to (replaced by) `ar′` and `i` to `i′`. Patterns may contain multiple terms (e.g. `{slot(ar,i),slot(ar,k)}` in line 5), meaning that the quantifier is instantiated only if the E-graph contains a matching term for *each* of these pattern terms. It is also possible to specify multiple (alternative) patterns for the same quantifier.

The choice of patterns is critical to the behaviour of the SMT solver. Since quantifier instantiations will *only* be considered when matching terms are encountered, overly restrictive patterns cause relevant quantifier instantiations to be missed (in the extreme case, if *no* ground term matching the pattern is encountered, the solver will behave as if the quantified formula were not present). But overly *permissive* patterns can cause too many quantifier instantiations, resulting in bad performance and even non-termination. The example in Fig. 1 performs around 5000 quantifier instantiations before *unknown* is reported, indicating that the solver can neither deduce unsatisfiability, nor confirm that its candidate model is correct (the solver cannot be certain whether the candidate model could be ruled out by extra quantifier instantiations not allowed by the patterns).

Why are so many quantifier instantiations made, for such a simple problem? One issue is the quantifier labelled Q_{nxt}, with a matching term `slot(ar′,i′)`. The resulting instantiation yields the assertion `next(slot(ar′,i′)) = slot(ar′,i′+1)`, in which the (new) ground term `slot(ar′,i′+1)` occurs; when added to the E-graph, this will trigger a new match, in a sequence which can continue

indefinitely (the solver terminates only because we bound the depth to 100). Such a repeating instantiation pattern is called a *matching loop*, and is a key cause of poorly performing solver runs. We will show in the next sections how we can systematically discover this matching loop and other quantifier-related problems in our example.

As illustrated by the quantifier Q_{srt}, terms in patterns may include nested function applications. Matching of ground terms against such patterns is not purely syntactic, but is performed modulo the *equalities* in the candidate model. For example, adding `x = slot(a,3)` $\wedge$ `lookup(h, x) = 42` to the example will trigger a match against the pattern `{lookup(h, slot(a,j))}`. The application of `lookup` can be rewritten via the assumed equality to `lookup(h, slot(a,3))`, which matches the pattern. Thus, understanding an instantiation requires knowledge not only of the available terms, but also of the equalities derived by the solver.

3 A Debugging Recipe for SMT Quantifiers

Even with input problems as simple as that of Fig. 1, undesirable quantifier instantiations easily occur; realistic problems generated by, for instance, program verification tools typically include many hundreds of quantifiers, thousands of terms, and a complex mixture of propositional and theory-specific constraints. Diagnosing and understanding performance problems is further complicated by the fact that the observed slow-downs may not be due to the quantifier instantiations alone; quantifier instantiations may generate many additional theory-specific terms which slow theory reasoning in the SMT solver, and disjunctive formulas which slow the case-splitting boolean search.

In order to systematically understand a quantifier-related SMT problem, we identify (based on our experience) the following sequence of debugging questions:

1. *Are there suspicious numbers of quantifier instantiations?* If not, poor performance is due to other causes, such as non-linear arithmetic reasoning.
2. *Which quantifiers exist in the given SMT problem, and what are their patterns?* The answer to this question is crucial for the subsequent steps and is by no means trivial: in many SMT applications, some quantifiers may be generated by client tools, the SMT solver itself may preprocess input formulas heavily (and heuristically select missing patterns), and nested quantifiers may be added only when outer quantifiers are instantiated.
3. *Which quantifiers are instantiated many times?* Our experience shows that most quantifier instantiation problems are caused by relatively few quantifiers. The quantifiers identified here will be further examined in the next steps.
4. To identify problematic quantifiers, it is often useful to explore the interactions between several quantifiers by asking:
   (a) *Does the causal relationship between quantifier instantiations exhibit high branching: that is, a single quantifier instantiation leads directly to many subsequent instantiations?* A typical example is when an instantiation produces new terms that lead to a combinatorial explosion of matches for another quantifier. Once we have identified such a situation, we analyse the involved quantifiers according to step 5.

(b) *Are there long sequences of instantiations causing one another?* Long sequences often indicate matching loops. To determine whether that's the case, we ask: *Is there a repeating sequence which indicates a matching loop?* If so, we analyse the involved quantifiers (as described in step 5) to determine whether and how this sequence can repeat indefinitely.

5. Once we have identified potentially problematic quantifiers, we analyse their individual instantiations by asking:
   (a) *Which pattern of the instantiated quantifier is matched, and to which terms?* The answer is needed to understand the cause of the instantiation, and particularly for identifying overly-permissive matching patterns.
   (b) *What do these terms mean with respect to the input problem?* SMT terms can often get very large and, thus, difficult to understand; tracing them back to the original problems facilitates the analysis.
   (c) *Is the match triggered via equality reasoning? Where do the necessary terms and equalities originate from?* Such matches are difficult to detect by manually inspecting the input problem because the patterns and the matching terms look syntactically different; instantiation problems that involve equality reasoning are especially difficult to debug by hand.

Except for the very first step in this recipe, efficiently answering these questions is impractical without tool support; our Axiom Profiler now provides this support.

4 Visualising Quantifier Instantiations

The Axiom Profiler takes as input a log file produced by Z3 and provides a wide range of features for analysing the performed quantifier instantiations. In this section, we show the key features for visualising and navigating the data from the log file. In the subsequent sections, we demonstrate how to analyse quantifier instantiation problems, both manually and automatically.

Figure 2 shows a screenshot for the example from Fig. 1. The tool shows in the middle panel raw data on quantifier instantiations from the log file plus some summary statistics, in the right-hand panel an instantiation graph with causal relationships between instantiations, and in the left-hand panel details of selected instantiations. We describe the three panels in the following.

Raw Data. The middle panel displays the raw data on quantifier instantiations, organised per quantifier as an (XML-like) hierarchy of fields; we inherited this view from the VCC Axiom Profiler [16]. The top-level statistics are useful as an overview for steps 1–3 of our debugging recipe (Sect. 3). Each line corresponds to an individual quantifier and shows its total number of instantiations ("`#instances`"). Manually navigating the underlying raw data is possible, but typically impractical.

In our example 11 quantifiers are listed: the first 3 are from our input problem; the remaining 8 are generated internally by Z3 (they are never instantiated, and we ignore them for our discussion). We can see there are more than 5000 quantifier instantiations in total; all three quantifiers are instantiated many times.

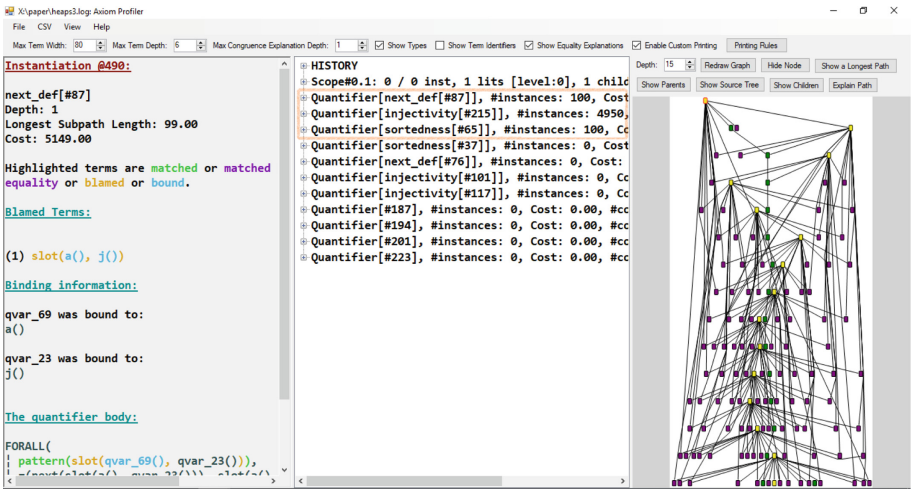

Fig. 2. A visualisation of the quantifier instantiations for the example in Fig. 1. (Colour figure online)

Instantiation Graph. The right-hand panel is one of the most important features of our tool. The *instantiation graph* visualises *causal relationships* between quantifier instantiations, which allows us to identify the high-branching and long-sequence scenarios described in step 4 of our debugging recipe. The nodes in the graph represent quantifier instantiations; a (directed) edge indicates that the source node provides either a term which was matched in order to trigger the successor node, or an equality used to trigger the match. Information about equalities is important for step 5c of our recipe, as we will discuss in Sect. 5.

Graph nodes are coloured, where each colour indicates a different quantifier; all instantiations of the same quantifier are coloured identically. The colours make it easy to spot prevalent quantifiers, and to visually identify patterns such as repeating sequences down a long path, which often indicate a matching loop.

Since instantiation graphs can get very large, we provide various filters, for instance, to control the maximum depth to which the graph is displayed, or to expand and collapse the children of a node. It is also possible to display the nodes with the highest number of children (to detect high branching, see step 4a) or the nodes starting the longest paths, as well as longest path starting from a node, which are features especially useful for detecting matching loops.

Our running example contains several instantiation problems. For instance, the instantiation graph in Fig. 2 shows a very large number of purple nodes (the quantifier labelled Q_{inj} in Fig. 1) and two sequences of yellow (the Q_{nxt} quantifier) and green (Q_{srt}) nodes, which indicate potential matching loops. Whereas the *number* of instantiations is also visible from the raw data, identifying such groupings and patterns is made possible by the instantiation graph.

Instantiation Details. The raw data and instantiation graph allow one to identify potentially-problematic quantifiers according to the first four steps of our debugging recipe. To support the analysis described in step 5, the left-hand panel provides details of all relevant information about specific quantifier instantiations. The instantiation of interest can be selected in either of the other two panels.

Selecting the top node in the graph from our example yields an explanation of the first instantiation of the (Q_{nxt}) quantifier. The panel lists *blamed terms*: that is, terms in the E-graph whose subterms were matched against the patterns; here, the blamed term is `slot(a,j)` (the numbers in square brackets are explained below). The subterm of a blamed term matched against the pattern (here, the whole term) is highlighted in gold, while the nested subterms bound to quantified variables (here, `a` and `j`) are shown in blue. The panel then shows the bindings per quantified variable (named by a unique number prefixed with "`qvar_`"), the quantifier itself (highlighting the pattern matched against), and any new resulting terms added to the E-graph. In particular, the bound terms and pattern matched against provide the information needed for step 5a of the debugging recipe.

In realistic examples, presenting the relevant terms readably can be a challenge, for which we provide a variety of features. Since the E-graph often contains only partial information about interpreted literals, it can be useful to annotate function applications and constants with a numeric *term identifier* (shown in square brackets); these identifiers are generated by Z3. For example, all integer-typed terms are simply represented by `Int()` here; the term identifiers allow us to identify identical terms, even though their precise meanings are unknown. Since identifiers can also make large terms harder to read, enabling them is optional.

For some problems, the relevant terms can get extremely large. To present terms in a meaningful form (see step 5b of our recipe), our tool provides facilities for defining custom printing rules. Typical use cases include simplifying names, and rendering those representing operators (such as a list append function) with infix syntax. In addition, our tool allows one to choose the depth to which terms are printed; we use `...` to replace subterms below this depth.

5 Manual Analysis of Instantiation Problems

In this section, we demonstrate on the example from Fig. 1 how to use the features of the Axiom Profiler to manually analyse and debug quantifier instantiation problems. The *automatic* analyses provided by our tool will be discussed in Sect. 6.

Simple Matching Loops. Since all three quantifiers in our example lead to many instantiations, we start narrowing down the problem by looking for matching loops (step 4b in the recipe). Filtering for nodes starting the longest paths displays the sub-graph on the left of Fig. 3. One can see two parallel sequences of instantiations; an initial one of yellow (Q_{nxt}) instantiations, and a second of green (Q_{srt}) instantiations. Since the (Q_{nxt}) sequence is self-contained (there are

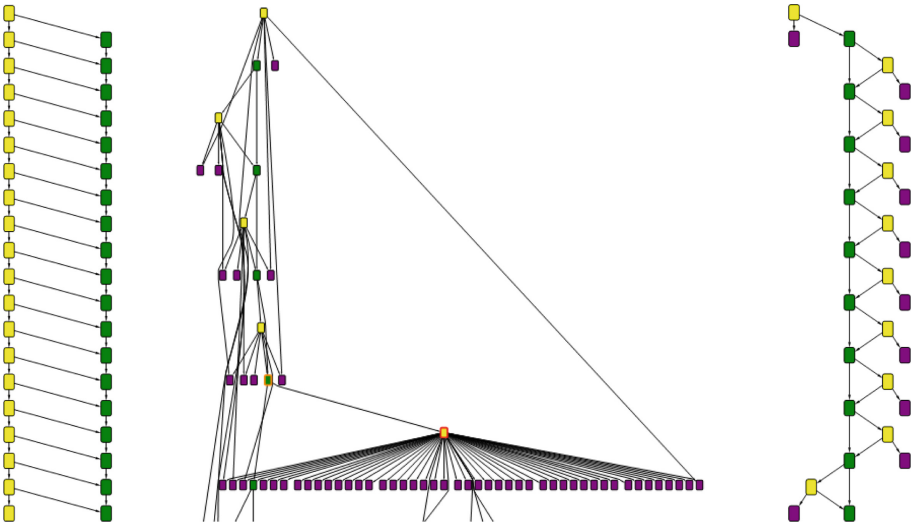

Fig. 3. Sub-graphs of the instantiation graph for the example in Fig. 1 showing the simple matching loop, high branching, and the matching loop with equality reasoning. Instantiations of Q_{nxt} are yellow, Q_{srt} is green, and Q_{inj} is purple. (Colour figure online)

no incoming edges from outside of these nodes), we investigate this one first. By selecting and inspecting the details of these instantiations (in the left-hand panel) in turn, we see that all after the first look very similar: each blames a term of the shape `slot(a, Int() + j)` (`Int()` abstracts all constants from the integer theory) and produces (among others) a new term of this same shape. The term identifiers show that the new term from each instantiation is the one blamed by the next, indicating a matching loop. In Sect. 6, we will show how the Axiom Profiler can perform this entire analysis automatically.

The detected matching loop for Q_{nxt} is the one we discussed in Sect. 2. It can be fixed by selecting a more restrictive pattern for the Q_{nxt} quantifier, namely `{next(slot(a,i))}`, only allowing instantiations when the `next` function is applied to an array slot. In particular, instantiating the quantifier does not produce a new term of this shape, which breaks the matching loop. Re-running Z3 on the fixed example reduces the number of quantifier instantiations to around 1400.

High Branching. Besides long paths, high branching may point to potentially problematic quantifiers (see step 4a of our debugging recipe). Once we have fixed the matching loop described above, we use the "Most Children" filter (available from "Redraw Graph") to identify the nodes with highest branching factor; subsequently using "Show Children" on one of these nodes results in the sub-graph in the middle of Fig. 3. This node has 42 child nodes, of which 41 are instantiations of the injectivity property (Q_{inj}). The pattern for this quantifier is `{slot(a,i),slot(a,k)}`, so each instantiation requires two applications

of the `slot` function. Examining the instantiation details reveals that the 41 instantiations all share one of the two `slot` terms, while the other varies. The common term is produced by the parent node, and then combined with many previously-existing terms to trigger the instantiation 41 times.

The underlying problem is that the pattern yields a number of instantiations that is quadratic in the number of `slot` terms. This is a known "anti-pattern" for expressing injectivity properties; an alternative is to add an inverse function and axioms expressing this fact [5], which can then match linearly in the number of `slot` terms. For simple injectivity axioms such as that from our example, Z3 will even perform this rewriting for us; we disabled this option for the sake of illustration. Enabling this reduces the number of quantifier instantiations to 152.

Equality Explanations. The remaining instantiations in our example form a long path, which may indicate another matching loop. As shown on the right of Fig. 3, the path alternates the quantifiers Q_{nxt} and Q_{srt}. Unlike for the simple matching loop we uncovered earlier in this section, neither of the two quantifiers now produces a term that *directly* matches the pattern for Q_{srt}. Instead, subsequent instantiations are triggered by rewriting terms via equalities.

The Axiom Profiler explains the required rewriting steps to support step 5c of the recipe. In particular, the instantiation details shown include, besides the blamed terms, the equalities used and how they can yield a term that matches the necessary pattern. In our example, the very first instantiation blames a term `lookup(h, next(slot(a, j)))` and shows the following relevant equality:

```
(2) next(slot(a, j))
    = (next_def[#56])
    slot(a, +(Int(), j))
```

where `(2)` is a number for this equality, and the `(next_def[#56])` annotation after the equality symbol is the *justification* for the equality; in this case, it names a quantifier (the Q_{nxt} quantifier in Fig. 1). In general, equality justifications can be more complex; we contributed code for Z3 to log relevant information, letting us reconstruct transitive equality steps, theory-generated equalities, and congruence closure steps (for which a recursive equality explanation can also be generated).

By inspecting each node's blamed terms and relevant equality information, it is possible to indeed uncover a matching loop still present in this version of our example. In brief: instantiating the Q_{srt} quantifier produces a term that triggers an instantiation of the Q_{nxt} quantifier, which produces equality (2). This equality is then used to rewrite the same term, resulting in another match for the Q_{srt} quantifier, and so on. Matching up all relevant terms and assembling the details of this explanation manually remains somewhat laborious; in the next section, we show a more detailed explanation which our tool produces *automatically*.

6 Automated Explanation of Matching Loops

The previous section illustrated how the Axiom Profiler supports the manual analysis of quantifier instantiation problems. For the common and severe problem of matching loops, our tool is also able to produce such explanations *automatically*, reducing the necessary debugging effort significantly.

Consider the example from Fig. 1, after fixing the first of the two matching loops as explained in the previous section. Recall that our *manual analysis* revealed that the second matching loop consists of repeated instantiations of quantifier Q_{srt}, which are sustained via equalities obtained from the quantifier Q_{nxt}. Applying our *automated analysis* produces the following explanation: (1) It identifies a potential matching loop involving the quantifiers Q_{nxt} and Q_{srt}. (2) It synthesises a template term `lookup(h, next(slot(a,`T_1`))` whose presence, for any term T_1, sets off the matching loop. (3) It explains step by step: (a) how such a term triggers the quantifier Q_{nxt} (recall that we fixed the pattern to `{next(slot(ar,i))}`) to produce the equality `next(slot(a,`T_1`))=slot(a,`T_1`+Int())`, (b) how this equality is used to rewrite the template term to `lookup(h, slot(a,`T_1`+Int())`, (c) that the resulting term causes an instantiation of quantifier Q_{srt} to produce the term `lookup(h, next(slot(a,`T_1`+Int()))`, and (d) how this term sets off the next iteration by using T_1`+Int()` for T_1. Our algorithm to produce such explanations consists of four main steps, which we explain in the remainder of this section.

Step 1: Selecting Paths. Our algorithm starts by selecting a path through the instantiation graph that represents a likely matching loop. The user can influence this choice by selecting a node that must be on the path and by selecting a sub-graph that must contain the path. The algorithm then chooses a path heuristically, favouring long paths and paths with many instantiations per quantifier (more precisely, per matched pattern). Since it is common that paths contain several instantiations before actually entering a matching loop, our algorithm prunes path prefixes if their quantifiers do not occur frequently in the rest of the path.

Step 2: Identifying Repeating Sequences. Matching loops cause repeated sequences of quantifier instantiations. We abstract the instantiations on a path to a *string* (instantiations of the same quantifier and pattern get the same character), and efficiently find the substring (subsequence of instantiations) repeating *most often* using suffix trees [23]; in our example, this subsequence is Q_{nxt}, Q_{srt}.

Step 3: Generalising Repetitions. Each repetition of the subsequence identified in the previous step potentially represents an iteration of a matching loop. To produce a generalised explanation for the entire loop, we first produce explanations of each individual repetition and then generalise those explanations.

The automatic explanation of the individual repetitions works as summarised in steps 5a and 5c of our debugging recipe, and uses the same functionality that we used for the manual analysis in the previous section. In our example, the analysis reveals that the first repetition of the sequence is triggered by the term

`lookup(h,slot(a,i))`, which comes from the assertion in line 13. This term triggers an instantiation of the quantifier Q_{srt}, which in turn triggers the quantifier Q_{nxt} to produce the equality `next(slot(a,i))=next(slot(a,i+1))`. Rewriting the term `lookup(h,next(slot(a,i)))` from the quantifier Q_{srt} with this equality produces `lookup(h,slot(a,i+1))`. Performing this analysis on the second repetition shows that it is triggered by exactly this term and, after the analogous sequence of steps, produces `lookup(h,slot(a,i+2))`, and so on.

The explanations for the individual repetitions of the sequence will differ in the exact terms they blame and equalities they use. However, since each repetition triggers the same patterns of the same quantifiers, these terms and equalities have term structure in common. We extract this common structure by performing anti-unification [19], that is, by replacing each position in which subterms *disagree* with fresh symbolic variables. In our example, anti-unification of the blamed terms for the first instantiation of each repetition produces `lookup(h,slot(a,`T_1`))`, that is, the disagreeing terms `i`, `i+1`, etc. have been replaced by the fresh symbolic variable T_1. Similarly, the used equalities are anti-unified to `next(slot(a,`T_2`))=next(slot(a,`T_3`+Int()))`.

Introducing a fresh symbolic variable in *each* such position loses information for terms originally occurring multiple times. For instance, in our example, anti-unifying the blamed terms and the used equalities introduces three symbolic variables T_1, T_2, T_3 even though the disagreeing terms are equal in each repetition of the sequence. This equality is vital for the explanation of the matching loop.

In simple examples such as this one, we need only keep the first introduced symbolic variable T; in general there may be different choices which can mutually express each other via function applications (e.g. $T = f(T')$ and $T' = g(T)$, for which *either* symbolic variable would be sufficient). We handle the general problem of selecting which symbolic variables to keep by building a directed graph to represent this expressibility relation between symbolic variables. We have developed an algorithm to efficiently select *some* subset of the symbolic variables with no redundant elements based on this graph; we then rewrite all generalised terms and equalities using only these variables. In our example, the graph reflects $T_1 \rightleftarrows T_2 \rightleftarrows T_3$ and, thus, we are able to rewrite the generalised equality to use only T_1 resulting in `next(slot(a,`T_1`))=next(slot(a,`T_1`+Int()))`.

Step 4: Characterising Matching Loops. Once we have generalised templates of the blamed terms and equalities, we use these to express the terms used to begin the *next* iteration of the repeating pattern; if this is a term with *additional* structure, we classify the overall path explained as a matching loop. In our example, we see that where T_1 was used, T_1`+Int()` is used in the next iteration, from which we conclude that this is indeed a matching loop. We add the information about these terms used to start the *next* iteration of the loop to our finalised explanations (*cf.* (d) in the explanation starting this section).

7 Implementation and Evaluation

Our work is implemented as a stand-alone application. We also submitted (accepted) patches to Z3 to obtain the full logging information that our tool requires; we now record equalities used for matching patterns, and justifications for *how* these equalities were derived from the solver's E-graph. These logging mechanisms are enabled via the additional `trace=true proof=true` logging options. Other SMT solvers could produce the same information (information on terms, equalities and matches), and reuse our work.

Example	Tool	#quants	#instantiations	#loops	longest	= used
compiler	Why3	1,473	195,961	1	12	
kmp	Why3	35	1,376	1	18	
blocking_semantics5	Why3	86	210,291	2	10	
fibonacci	Why3	234	32,647	2	19	
induction	Why3	2	197	1	20	
sf	Why3	5	2,020	2	16	
sumrange	Why3	10	1,837	2	19	
vstte10_queens	Why3	16	194	1	18	
unionfind	Viper	98	285,311	1	100	✓
linked_list_qp_append	Viper	196	17,470	1	96	
testHistoryLemmasPVL	Viper	4	271	2	100	✓
testHistoryThreadsLemmasPVL	Viper	4	270	2	100	✓
tree_delete_min	Viper	19	6,709	1	96	✓
list_insert	Viper	24	287,559	1	94	✓
list_insert_heuristics	Viper	23	181,392	1	94	✓
tree_delete_min_heuristics	Viper	19	29,747	1	96	✓
tree_delete_min_no_assert	Viper	19	120,776	1	98	✓
ComputationsLoop	Dafny	16	518	1	33	
ComputationsLoop2	Dafny	20	519	1	33	
NoTypeArgs	Dafny	59	40,110	1	98	✓
Lucas-down	Dafny	86	3,412	1	99	
Lucas-up	Dafny	129	46,877	1	91	

Fig. 4. An overview of the matching loops found in examples flagged by the automated analysis. "#quants" indicated the no. of quantifiers present (we only count those instantiated at least once). "longest" indicates the length of the longest path; "= used" indicates whether equality explanations occur in the generalised path explanation for this matching loop. Examples exhibiting similar matching loops are grouped together.

In order to demonstrate that the Axiom Profiler can be used to identify and help explain matching loops in real world problems, we ran experiments on a total of 34,159 SMT files, corresponding to the full test suites of a selection of verification tools which use Z3 as a backend: F* [22] (6,281 files), Why3 [10] (26,258 files), Viper (887 files) [18], Nagini [9] (266 files) and Dafny [13] (467 files)). Using a command-line interface for our tool, we analysed each of these files in an attempt to find matching loops. We note that since these are expert-written, polished (and, presumably performant) test suite examples, one might expect most, if not all such issues to have been eliminated.

For each file, the tool analysed the 40 longest paths in the instantiation graph, searching for matching loops using the analysis of Sect. 6. In order to eliminate most false positives (since paths were found with no user introspection, and repeating patterns do not always indicate matching loops), we searched for paths with at least *10* repetitions of some quantifier instantiation sequence. We found 51 such files, and inspected each by hand for matching loops, using the techniques of Sects. 4 and 6 to identify whether matching loops are present (false positives also occurred: for example, in tools which model lookups in program heaps using axioms, there are often long chains of instantiations of the same axiom but for different versions of the heap). For all investigated examples, our tool analyses these paths in a second or two: this manual classification is very efficient.

Figure 4 summarises the matching loops we discovered. We found 28 previously-unknown matching loops: 13 from Why3, 10 from Viper and 5 from Dafny; we didn't find any matching loops in the F* or Nagini suites (which could be due to highly-tuned or restrictive triggers). The matching loops we detected were of many varieties. In Why3, 10 stemmed from modelling recursively-defined concepts (e.g. ∀`x: Int even(x). even(x)` ⟹ `even(x + 2)`). We also found more complex matching loops in Why3's axiomatization of lists and arrays. For example, an array axiom ∀`x, y. {mk_ref(x,y)}. sort(ref(x),mk_ref(x,y))` yields a term matching a second axiom, yielding `contents(ref(`T_1`), mk_ref(ref(`T_1`), mk_ref(`T_1`, `T_2`)))`. The outer `mk_ref` term is new; we can instantiate the first axiom again. Why3 leaves the selection of most patterns to the SMT-solver, and uses a timeout (along with alternative solvers); this explains the potential for matching loops.

Viper provides two verifiers; we extracted and ran SMT queries for both. We found 4 causes of matching loops; some manifested in multiple files. These include direct matching loops and more complex cases: e.g. in the `testHistoryLemmasPVL` example, an associativity axiom allows repeatedly generating new term structure, which feeds a parallel matching loop concerning a recursive function definition.

For Dafny, we also found some simple and some complex cases. One axiom ∀`a, b, c {app(a, app(b, c))}. app(a, app(b, c)) = app(app(a, b), c)` expressing associativity of a concatenation operator[3], in combination with a case-split assumption made by Z3 that one term `a'` instantiated for `a` is equal to `Nil`, and the known property `Nil=app(Nil,Nil)`, allows rewriting the right-hand-side term learned from each instantiation with a new `app` application on which the same axiom can be matched. A similar problem is described by Moskal [16].

In all cases (including those which, when inspected manually turned out to be false positives), following our debugging recipe and applying the Axiom Profiler's features allowed us to quickly isolate and explain the matching loops present. We have communicated our findings to the respective tool authors, who all confirmed that these matching loops were previously-unknown and that they plan to investigate them further (potentially using the Axiom Profiler itself).

[3] The actual function name is `concat`; we abbreviate for readability.

8 Related Work

Since its origin in the Simplify prover [7], E-matching has been adapted and improved in implementations for a variety of SMT solvers [1,2,4,11,17]. Since E-matching-based instantiation gives weak guarantees for *satisfiable* problems (typically returning *unknown* as an outcome), for problem domains where satisfiability (and a corresponding model) is the desired outcome, alternative instantiation techniques have been proposed [12,20,21]. For specific domains, these are often preferable, but for problems in which many external concepts need to be modelled with quantifiers, such as deductive program verification, E-matching remains the only general solution. While our work focuses on E-matching support, it would be interesting future work to investigate to what extent we could also provide useful information about other quantifier instantiation strategies.

As discussed in the Introduction, we build upon the *VCC Axiom Profiler* [16] tool, which defined first versions of the logging in Z3 (without equality information), the raw data display (retained in our middle panel) and a basic display of information per quantifier, without explanations of equalities used to justify matches. The contributions of this paper make it practical to quickly navigate and understand even complicated SMT runs, in ways impossible with the previous tool. Nonetheless, this prior tool was a very helpful basis for our implementation.

The serious challenges of pattern selection have warranted papers both on expert strategies [14,16], and for formalising the logical meaning of quantifiers equipped with patterns [8]. Various SMT solvers select patterns for quantifiers automatically (if omitted by the user). To reduce the uncertainty introduced in this way, many program verification tools select their own patterns when encoding to SMT (e.g., VCC [3], Viper [18], Dafny [13]). Leino and Pit-Claudel [15] present a technique for selecting patterns in Dafny while avoiding direct matching loops; the matching loops we found in Dafny tests arose in spite of this functionality.

9 Conclusions

In this paper, we presented a comprehensive solution for the analysis of quantifier instantiations in SMT solvers. Our newly-developed Axiom Profiler enables a user to effectively explore and understand the quantifier instantiations performed by an SMT run, their connections and potentially-problematic patterns which arise (e.g. due to matching loops). Our instantiation graph, customisable visualisation of information and automatic explanations for matching loops make investigating even complex SMT queries practical in reasonable time. Furthermore, we were able to script these analyses to uncover matching loops in a variety of test suites for existing tools; it would be interesting to analyse further tools in this way.

As future work, we plan to investigate tighter integration with tools that build on SMT solvers, e.g. to represent terms at a higher level of abstraction. We also plan to investigate whether theory-reasoning steps in the SMT solver can

be made less opaque to our tool, especially with respect to justifying equalities. Automating explanations for matching loops with repeating structures more complex than single paths would be a challenging extension of our techniques.

Acknowledgements. We thank Frederik Rothenberger for his substantial work on visualisation features. We are grateful to Marco Eilers, Jean-Christophe Filliâtre, Rustan Leino, Nikhil Swamy for providing their test suites and advice on their verification tools. We thank Nikolaj Bjørner for his assistance with Z3, and Michał Moskal for generous advice and feedback on earlier versions of the tool. Finally, we are very grateful to Marco Eilers, Malte Schwerhoff and Arshavir Ter-Gabrielyan, for providing extensive feedback on our tool and paper drafts.

References

1. Bansal, K., Reynolds, A., King, T., Barrett, C., Wies, T.: Deciding local theory extensions via E-matching. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part II. LNCS, vol. 9207, pp. 87–105. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_6
2. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
4. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
5. de Moura, L., Bjørner, N.: Z3 - a tutorial. Technical report, Microsoft Research (2010)
6. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
8. Dross, C., Conchon, S., Kanig, J., Paskevich, A.: Reasoning with triggers. In: Fontaine, P., Goel, A. (eds.) Satisfiability Modulo Theories (SMT). EPiC Series in Computing, vol. 20, pp. 22–31. EasyChair (2012)
9. Eilers, M., Müller, P.: Nagini: a static verifier for Python. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 596–603. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_33
10. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
11. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 167–182. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_12

12. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
13. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
14. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009, pp. 615–622. ACM, New York (2009)
15. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part I. LNCS, vol. 9779, pp. 361–381. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_20
16. Moskal, M.: Programming with triggers. In: SMT. ACM International Conference Proceeding Series, vol. 375, pp. 20–29. ACM (2009)
17. Moskal, M., Lopuszański, J., Kiniry, J.R.: E-matching for fun and profit. Electron. Notes Theor. Comput. Sci. **198**(2), 19–35 (2008)
18. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
19. Plotkin, G.D.: A note on inductive generalization. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, pp. 153–163. Edinburgh University Press, Edinburgh (1970)
20. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in SMT. In: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design, FMCAD 2014, pp. 31:195–31:202. FMCAD Inc., Austin (2014)
21. Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 377–391. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_26
22. Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, pp. 266–278. ACM, New York (2011)
23. Weiner, P.: Linear pattern matching algorithms. In: Switching and Automata Theory (SWAT), pp. 1–11. IEEE (1973)