Developing Tool Support for Understanding Quantifier Instantiations

Bachelor's Thesis Project Description

Oskari Jyrkinen
Supervisors: Prof. Peter Müller, Jonas Fiala, Dionisios Spiliopoulos
Department of Computer Science
ETH Zürich
Zürich, Switzerland

September 2023

1 Introduction

Program verifiers, such as Viper [5], rely on SMT solvers to prove the validity of logical formulas. Checking satisfiability with SAT solvers is decidable for logical formulas that only involve propositional logic. However, propositional logic is not sufficiently expressive for many problems in program verification and hence we often need predicate logic, in particular formulas involving universal quantifiers. But since the satisfiability of SMT assertions with quantifiers is undecidable in general [2], SMT solvers resort to heuristic approaches, such as *E-matching* [4].

An *E-graph* is a data structure for representing equivalences among terms. During E-matching, the SMT solver constructs an E-graph consisting of the *ground terms*, i.e. terms without bound variables, of the formulas and then repeatedly instantiates the quantified formulas to gain new facts with which it then tries to prove the unsatisfiability of a formula. The SMT solver uses *triggers* to find expressions in the E-graph that have a matching pattern such that it can bind the bound variables of the quantified formula to some ground terms and hence instantiate the quantified formula. Triggers are expressions that include the quantified variables of the formula and they are either user-specified or generated automatically [6].

A commonly encountered problem in this context is that the solver can suffer from poor performance, e.g. due to *matching loops*. A case where this can occur is if we have a quantified formula with some trigger that matches a term in the E-graph. The SMT solver might instantiate that formula to obtain a new fact consisting of some terms that are added to the E-graph. If the newly obtained terms again contain terms that can be matched to the trigger of the same quantified formula, we can again instantiate it to obtain new facts that again contain new terms matching with the trigger and hence the loop continues. Termination can be ensured with an instantiation threshold, however, the performance can suffer nevertheless. In some cases matching loops are necessary to prove certain assertions. Hence, if we have a too restrictive instantiation threshold, the SMT solver might be unable to prove an assertion.

Since a large fraction of program verification time is consumed by the SMT solver, it makes sense to gain more insight into this step. All of the aforementioned problems clearly indicate that it is desirable to have tool-support for gaining insight into how SMT solvers instantiate the formulas during a solver run.

2 Motivation

As described in the previous section, there are issues associated with *E-matching* that can lead to performance degradation. These include matching loops and *high branching* as described by Becker et al. [2]. Let's make an example of matching loops to obtain a better understanding.

```
1 (declare-fun slot(Arr Int) Loc)
2 (declare-fun lookup(Heap Loc) Int)
3 (declare-fun next(Loc) Loc)
4 (assert ∀ar:Arr, i:Int ::  {slot(ar,i)} next(slot(ar,i)) = slot(ar,i+1)) ; Q_nxt
5 (declare-const h Heap)
6 (declare-const a Arr)
7 (declare-const j Int)
8 (assert (not (lookup(h, slot(a,j)) > (lookup(h, next(slot(a,j)))))))
```

In line 8 we can see that the term `slot(a,j)` is a ground term since it does not contain any quantified variables. Note that in lines 6-7, `a` and `j` are declared as constants. Therefore, this term will occur in the E-graph. Unless the assertions have already been proven or the limit for the instantiation depth has already been reached, the solver will match the trigger `slot(ar,i)` at line 4 to `slot(a,j)` and hence instantiate the formula Q_{nxt} with `ar=a` and `i=j` such that we obtain the new fact `next(slot(a,j)) = slot(a,j+1)`. Note that this equality contains the ground term `slot(a,j+1)` which is hence added to the E-graph. Since this new term again matches with the trigger `slot(ar,i)`, the corresponding formula can again be instantiated but this time with `ar=a` and `i=j+1` to obtain the new fact `next(slot(a,j+1)) = slot(a,(j+1)+1)`. The same reasoning can now again be applied to show that we can again match the same trigger to the new terms and hence we can continue with this until the SMT solver has reached the limit for the instantiation depth.

During the run of an SMT solver, such as Z3 [3], log files can be generated that contain information about how quantifiers were instantiated during the solver run. This allows us to learn for each instantiation of a quantified formula, which formula was instantiated, which terms of the E-graph we are matching against, how the quantified variables were instantiated and what resulting terms were obtained. This motivates the definition of an *instantiation graph* where each node Q corresponds to an instantiation of a quantified formula and there is an edge (Q_1, Q_2) between two nodes, Q_1, Q_2, if the matched term of Q_2 was a resulting term of the instantiation of Q_1, as in the example of Figure 1.

If we have a long path of nodes all of which correspond to Q_{nxt}, we could deduce upon closer inspection whether there is indeed an undesired matching loop or not and hence change the triggers

Formula	Matched term	Bound variables	Resulting terms
Q_{nxt}	`slot(a,j)`	`ar=a, i=j`	`next(slot(a,j))=slot(a,j+1)`
Q_{nxt}	`slot(a,j+1)`	`ar=a, i=j+1`	`next(slot(a,j+1))=slot(a,j+2)`
Q_{nxt}	`slot(a,j+2)`	`ar=a, i=j+2`	`next(slot(a,j+2))=slot(a,j+3)`
⋮	⋮	⋮	⋮

(a) Each row represents information on the instantiation of the formula specified in the left-most column.

$$\begin{tikzcd} \boxed{Q_{nxt}} \arrow[r] & \boxed{Q_{nxt}} \arrow[r] & \boxed{Q_{nxt}} \arrow[r] & \cdots \end{tikzcd}$$

(b) Instantiation graph: Each node represents an instantiation and hence a row in the table above. The leftmost node represents the instantiation in the first row of the table and the middle node represents the instantiation in the second row of the table. The reason why there is an edge from the first to the second node from the left is because the matched term of the second instantiation, `slot(a,j+1)`, is contained in the resulting terms of the first instantiation.

Figure 1: Pictorial representation of a matching loop.

or the formulas in such a way that the matching loop does not occur. In other cases, manual inspection of the instantiation graph could allow us to deduce that the instantiation threshold has been reached before the solver was able to prove an assertion and hence that the instantiation threshold should be increased. Given that even for rather small verification problems the log files can contain millions of lines of verbose text that is hard to read, tool-support is clearly necessary. Furthermore, constructing such an instantiation graph from a log file could potentially also allow for automatic detection of matching loops.

3 Goals

3.1 Core Goals

3.1.1 Tool for Analyzing SMT Solver Log Files

The goal is to create a web application that takes as input a log file produced by a solver run of an SMT solver (Z3) and

- can specify for each instantiated quantifier
  - which quantified formula was instantiated
  - which term of the E-graph the trigger was matched against
  - how the quantified variables were instantiated
  - the resulting terms obtained from the instantiation
- can explain how terms were rewritten due to equivalences between the terms in the E-graph

- can visualise the instantiation graph
- can export the visualisation in a format that is suitable for use outside of the tool itself (e.g., SVG)

3.1.2 Usability Testing

There have been previous efforts to build similar tools [1] and one core goal of this project is to make this tool more user-friendly and easily accessible by turning it into a web application. To validate user-friendliness, another core goal should hence be to conduct some form of usability testing. This would allow us to obtain valuable feedback that we could incorporate into the tool.

3.1.3 Find and Implement Additional Useful Functionality

It is not entirely evident whether the suggested visualisation tool is sufficiently insightful for users. Hence we want to figure out using real-world examples and/or feedback from users what kind of features or visualisations are desirable for gaining more insight into the internals of the SMT solver and then to implement that. Unless we find something better, it might be sensible to visualise the ground terms in a way that allows users to infer whether the SMT solver is doing lots of quantifier instantiations before reaching the threshold or whether the SMT solver is quickly reaching the instantiation threshold.

3.2 Extension Goals

3.2.1 Automatic Matching Loop Detection

Ideally, the tool should help users detect matching loops. This would allow users to more quickly identify their root cause and hence adjust the involved formulas and/or triggers to avoid matching loops and hence improve performance.

3.2.2 Mathematical Definition of Matching Loops

In order to create an automatic loop detection, it's necessary to find a mathematical definition of matching loops such that we can more easily reason about the correctness of a matching loop detection algorithm.

3.2.3 Real Time Visualisation

To allow the user to gain even more insight into a solver run of Z3, it would be convenient if the user could just enter an `.smt2` file into the browser and inspect how the quantifier instantiation graph is built during the solver run of Z3. This would allow users to gain insight about matching loops early on and not just at the end. Given that solver runs of Z3 can take quite long, this seems like a reasonable feature.

References

[1] Nils Becker. *Axiom Profiler.* URL: `https://github.com/viperproject/axiom-profiler`. (accessed: 24.02.2023).

[2] Nils Becker, Peter Müller, and Alexander J. Summers. “The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 99–116. ISBN: 978-3-030-17462-0.

[3] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14.* Springer. 2008, pp. 337–340.

[4] Leonardo De Moura and Nikolaj S Bjørner. “Efficient E-matching for SMT solvers”. In: *CADE.* Vol. 4603. Springer. 2007, pp. 183–198.

[5] Uri Juhasz et al. *Viper: A verification infrastructure for permission-based reasoning.* Tech. rep. ETH Zurich, 2014.

[6] K Rustan M Leino and Clément Pit-Claudel. “Trigger selection strategies to stabilize program verifiers”. In: *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28.* Springer. 2016, pp. 361–381.