User-Propagation for Custom Theories in SMT Solving

Nikolaj Bjørner[1], Clemens Eisenhofer[2,*] and Laura Kovács[2]

[1]Microsoft Research Lab, Microsoft Building 99, 14820 NE 36th Street, Redmond, Washington, 98052, USA
[2]TU Wien, Institut für Logic and Computation, Favoritenstraße 9-11, 1040 Wien, Austria

Abstract

We present ongoing work on developing a user-propagator framework in SMT solving. We argue that the integration of user-propagators in SMT solving yields an efficient approach towards custom theory reasoning, without bringing fundamental changes in the underlining SMT architecture. We showcase our approach in the SMT solver Z3, provide practical evidence of our work, and also discuss potential venues for further improvements.

Keywords

theory reasoning, lazy encoding, SMT callbacks

1. Introduction

Over the past years, domain-specific proving procedures have been proposed, for example, to solve constraints in non-linear arithmetic [1], string theory [2, 3, 4, 5], term algebras [6, 7], or bit-vectors [8, 9]. While it would be possible to provide pages with lists of heuristics and encoding practices in these and similar reasoning approaches over (first-order) theory constraints, the summary is as simple as follows: different theories need different reasoning approaches tailored to the respective theories. In this extended abstract we argue that by using so-called user-propagators as extensions to existing reasoning engines, in particular SMT solvers, we gain custom support for new theories without destroying the already efficient and highly-optimized reasoning infrastructure of the respective solvers. While in our work we focus on the efficient addition of user-propagators to the Z3 SMT solver [10], we believe our approach can be beneficial for improving theory reasoning in SMT solving in general.

In a nutshell, a user-propagator implements a set of functions, called callbacks, that are called whenever a potentially relevant action is performed by the solver. Intuitively, the callbacks of a user-propagator enable the on demand addition of theory formulas (lemmas) to the solver, which is especially important for theories that do not yet have a native reasoning support in the respective solver but support lazy clause generation [11, 12]. Thanks to on demand activation of theory lemmas, user-propagators help to restrict the search space of SMT solving with minimal overhead, as evidenced by our initial results (see Section 2). In this extended abstract we discuss

SMT 2022: Satisfiability Modulo Theories, August 11–12, 2022, Haifa, Israel
*Corresponding author.
nbjorner@microsoft.com (N. Bjørner); clemens.eisenhofer@tuwien.ac.at (C. Eisenhofer); laura.kovacs@tuwien.ac.at (L. Kovács)
0000-0002-1695-2810 (N. Bjørner); 0000-0003-0339-1580 (C. Eisenhofer); 0000-0002-8299-2714 (L. Kovács)

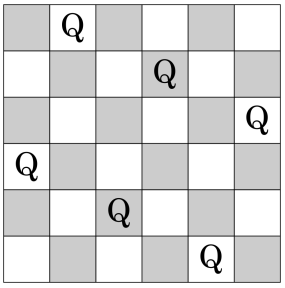

Figure 1: A valid placement of 6 queens in the 6-queens problem.

our approach towards a tailored integration of user-propagators with Z3 (Section 3), and outline ideas for further improvements (Section 4).

2. Motivating Example

We illustrate the benefits of user-propagators via solving the n-queens problem. Recall that the n-queens problem asks for a placement of n mutually non-threatening queens on an $n \times n$ chessboard, with $n \geq 1$. That is, the placement should ensure that none of the queens can attack any of the other ones, where queens may attack along rows, columns, and diagonals on the chessboards. An example of a placement solution for the n-queens problem, when $n = 6$, is shown in Figure 1.

One way to formalize the n-queens problem is to use (bit-vector) linear arithmetic, and assert a constraint problem (CP) as the following formula

$$distinct(q_1, \ldots, q_n) \wedge \bigwedge_{1 \leq i \leq n} q_i < n \wedge \bigwedge_{1 \leq i < j \leq n} |q_i - q_j| \neq |i - j|, \tag{1}$$

where q_i represents the numerical position of a queen in the i^{th} row. A solution to (1) gives a valid placement of n queens on the chessboard.

User-Propagator for Lazy Encoding in SMT. We note that SAT/SMT approaches over bit-vector arithmetic, such as [9, 10], can be used to solve (1), by reasoning in the full theory of bit-vector arithmetic. In this paper, we argue and show that implementing a user-propagator on top of SMT solving may outperform such approaches. The main benefit of a user-propagator comes with adding theory-constraints on demand; we therefore refer to SMT solving with user-propagators as *lazy encoding in SMT*, as also advocated in [11, 12]. In more detail, when using a user-propagator in the SMT solver to solve (1), the user-propagator tracks/watches the variable assignments made during SMT decisions, and adds conflicts to the SMT problem in case (1) is (partially) violated. This way, the formula (1) is solved by resolving the conflicts added by the user-propagator, avoiding the potential overhead of bit-blasting the whole formula. A pseudo code showing the general idea can be found in Listing 1.

Listing 1: Lazy Bit-Vector Encoding of n-Queens

```
fixed(ast, value) :

    queenY = queenToY(ast)
    queenX = value

    if (queenX ≥ board)
        // Constraint: queens must be on the board
        conflict({ ast })
        return

    foreach (fixed in alreadyFixedVars)
        otherX = model[fixed]
        otherY = queenToY(fixed)

        if (queenX = otherX)
            // Constraint: queens may not attack vertically
            conflict({ ast, fixed })
        else if (|queenX - otherX| = |queenY - otherY|)
            // Constraint: queens may not attack diagonally
            conflict({ ast, fixed })
```

Experimental Results. To showcase the practical benefits of user-propagators in SMT solving, we carried out experiments on generating *all solutions to the n-queens problem*[1]. That is, we generate all valid placements of n queens on an $n \times n$ chessboard. To this end, we used the following experimental setup:

(i) We used a standard SMT approach to solve and enumerate all solutions to the n-queens problem, by reasoning over (1) in bit-vector arithmetic. In this setting, the formula will be bit-blasted completely by the solver before the actual reasoning starts. We refer to this experiment as an *eager encoding with externally added conflicts*, as each solution is generated by a new SMT run (using an incremental solver). Once a solution is generated, the negation of this solution is used as a blocking clause to generate a new solution, via another SMT solving process.

(ii) When using a standard SMT approach as in (i) to generate all solutions to the n-queens problem, there is a significant burden in extracting and revising (new) models by generating and storing so-called blocking clauses (clauses that resolve decision conflicts), even if an incremental solver is used. To mitigate this burden, we added a conflict between all queen variables every time the SMT solver consistently fixed the last previously unassigned variable. This allows the SMT solver to enumerate all models within a single query, rather than enumerating solutions one-by-one as in case (i). The conflicts (i.e. blocking clauses) added through the user-propagator are considered as learned clauses, and can

[1] See https://github.com/Z3Prover/z3/tree/master/examples/userPropagator for the code.

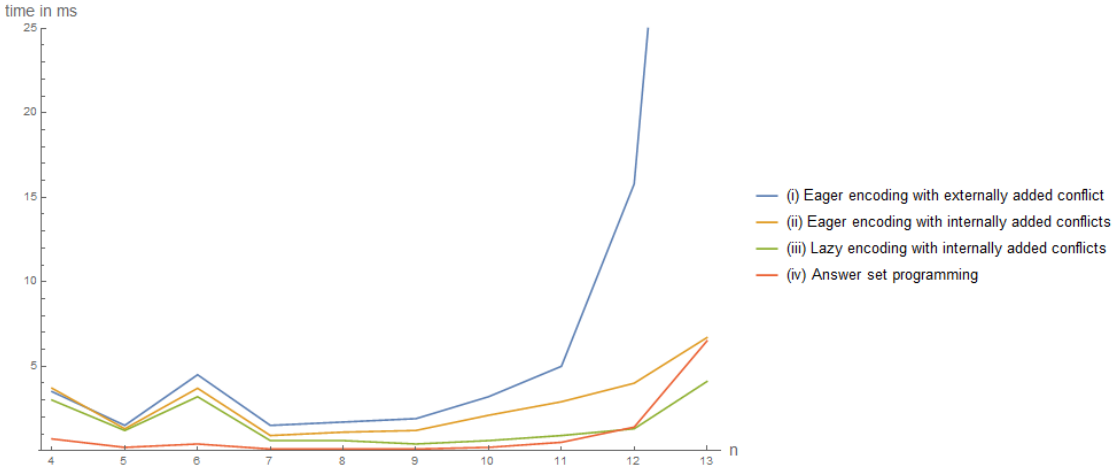

Figure 2: Running times for generating all solutions of the n-queens problems relative to the total number of solutions, using a bit-vector encoding.

be disposed by the solver. As clauses that are still relevant might be disposed as well, the user-propagator has to keep track of all solutions found so far and block them again in case required as the solver may find the same model multiple times. We refer to this experiment as an *eager encoding with internally added conflicts*.

(iii) We used a *user-propagator for lazy encoding in SMT solving*, as described above. That is, we on demand add theory-constraints generated by the user-propagator as blocking clauses to the SMT solver; in other words, we lazily instantiate theory-constraints using the user-propagator. We refer to this experiment as *lazy encoding with internally added conflicts*.

(iv) Finally, we used an *answer set programming* (ASP) approach [13, 14] to generate all solutions to the n-queens problem (1). We considered this experimental comparison, as answer set programming allows a similar syntactic encoding, and is considered to be efficient at enumerating all possible solutions to a given problem of a finite domain.

For performing our experiments, we used the SMT solver Z3 in (i)-(iii), and the ASP solver clingo [14] in (iv). Our experimental results from (i)-(iv) are plotted on Figure 2, showcasing that our user-propagator approach (iii) to lazy encoding in SMT outperforms the other settings. Although the ASP encoding is very fast compared to the other encodings for small n, it performs worse than (ii) and (iii) when $n > 13$ and $n > 12$, respectively.

We note that there are other more efficient encodings for the n-queens problem as well. For example, a direct translation to SAT is possible by stating that there has to be a queen in every row and manually forbidding all pairs of positions that cannot be both occupied at the same time by respective clauses. Our experimental results using this completely propositional encoding, together with *eager* and *lazy* encodings are summarized in Figure 3, showcasing again that user-propagation in SMT can be used to speed up reasoning. Although, finding all

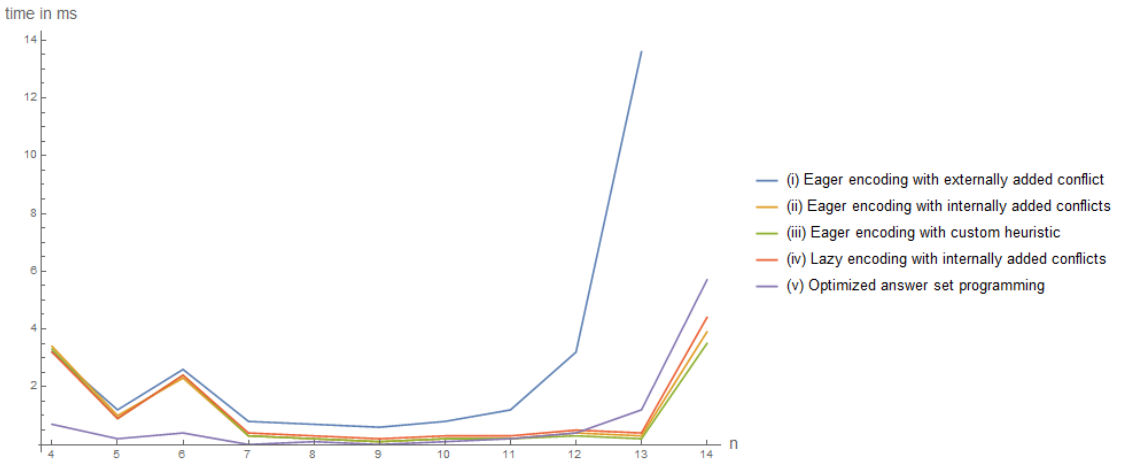

Figure 3: Running times for generating all solutions of the n-queens problems relative to the total number of solutions, using a propositional encoding.

solutions by this completely propositional approach shows that lazy encodings do not give a generic approach that decrease reasoning time in all cases, user-propagators can also be used to implement custom variable selection/assignment heuristic to improve performance. Our evaluation includes the results of using an eager propositional encoding together with a heuristic implemented through the user-propagator's. The heuristic first puts queens on tiles where they attack the most squares not previously attacked, possibly increasing the number of subsequent assignments that can be done by constraint propagation. An eager propositional encoding together with this heuristic turned out to be the most efficient variant.

Figure 4 shows that, although finding all solutions to the n-queens problem by a lazy propositional approach does not perform well, the same approach strongly reduces the time of finding a single solution. In addition, the required amount of memory has been greatly reduced: Finding a solution to $n = 60$ required 84 megabytes with the eager bit-vector encoding and 128 with the eager propositional one. The lazy encodings, however, both required only about 19 megabytes.

3. User-Propagator in SMT Solving with Z3

As mentioned, a user-propagator implements a set of callbacks, where callbacks are custom-defined by the user. When using a user-propagator, the CDCL($\mathcal{T}$) engine of the SMT solver will call the respective callback if some of the later described events occur, in order to allow the client to observe the solvers actions and interfere if required. To keep the overhead because of the additional callbacks low and to get only callbacks that are relevant for the user-propagator, the SMT solver tracks only events that are related to expressions that were previously registered by the user-propagator.

As shown in Section 2, user-propagators in SMT solving can be used to solve constraint

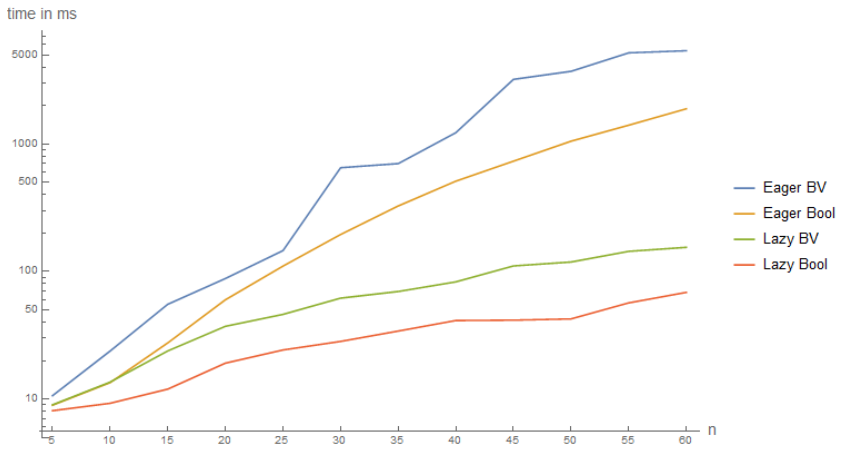

Figure 4: Running times (log. scaled) of finding a single solution for $n = 5, \ldots, 60$.

problems by offering a way to implement lazy clause generation [11, 12]. Moreover, proper implementations of user-propagators may significantly increase the efficiency of the solving process. This is mainly due to the fact that the SMT solver does not have to keep a large number of (potentially irrelevant) clauses in its memory, while tracking a specific variable assignment order, or by detecting invalid variable assignments early.

In what follows, we describe our approach to support user-propagator callbacks in SMT solving, by focusing on the direct integration of these callbacks with Z3. Our experiments showed that the integration comes with minimal overhead on the solver side, which can be easily compensated by its amenities. We note that user-propagators have already been supported in another form in Z3, by using theory plug-ins that allowed extending Z3's native theory support with so-called user-theories [15]. Based on this theory plug-in interface of Z3, a general purpose string solver was developed in [16]. Similar ideas for extending theory support in SMT have also been exploited in the SMT solver OpenSMT [17].

More recently, in [18] custom user propagators through the API of Z3 have been reintroduced for booleans and bit-vectors to solve a configuration problem in the context of large-scale constraint problems. The approach of [18] shows how manually introduced conflicts can be used to enforce some pseudo-boolean constraints during SMT solving. In extension of [18], in our work we employ user-propagators not only to solve satisfiability, but to also efficiently generate all models/solutions of a constraint problem by adequately integrating user-propagator callbacks in the SMT solving process. In particular, we support the following callbacks in the CDCL($\mathcal{T}$) engine of Z3:

Push and Pop – are invoked when Z3 branches (Push) on a boolean decision, respectively, backtracks (Pop).

Fixed – is invoked when a registered boolean/bit-vector expression is assigned a fixed value.

A bit-vector expression is fixed, when all involved bits are fixed.

Eq and Diseq – are invoked when two registered expressions are inferred equal (Eq) or disequal (Diseq). In contrast to the Eq callback, Diseq is incomplete, as it only reports the disequality if there is a formula in Z3's internal formula representation containing a respective equality atom that was set to false.

Final – is invoked when there are no more decisions to make.

Decide – is invoked when Z3 branches on a registered expression. The function may decline the chosen variable to branch on and its value by providing an alternative. Custom variable selection/assignment heuristics can be implemented through this callback.

Created – is invoked when an instance of a user-function is encountered the first time. This callback is especially relevant in case a function occurs within the scope of a quantifier; in this case, Z3 may instantiate the function several times with different arguments.

Fresh – is invoked when a solver creates a new sub-solver instance for doing subqueries. This is done, for example, in course of model based quantifier instantiation (MBQI).

We conclude by noting that most of the above callbacks may propagate arbitrary new formulas to the CDCL($\mathcal{T}$) engine of Z3, add conflicts between already fixed variables within Z3, or set the variable to split on next (similar to the decide-callback). For example, when generating all solutions of the n-queens problem in Section 2, we enforced some global constraints by lazy clause generation, while observing bit-vector assignments and introducing conflicts. As evidenced in Figures 2–4, we believe that a properly integrated user-propagator in SMT solving can strongly reduce memory consumption and reasoning time by adding theory constraints on demand.

4. Conclusion and Future Work

We discuss user-propagators in SMT solving with the aim of providing efficient and on demand theory-reasoning. Our initial experiments in this respect are encouraging. Beyond our motivating example using the n-queens problem, we are currently applying user-propagators in Z3 to improve performance of software verification tools, such as the alive2 verification framework [19]. In particular we try to delay axiom instantiations that are very unlikely to influence the SMT solver's outcome. For example, we want to ensure that different memory allocations yield disjoint memory addresses. Reasoning about such and similar properties, encoded in bit-vector arithmetic, yields a quadratic number of instances when eagerly instantiating bit-vector axioms. Instead, by using a user-propagator, we lazily instantiate bit-vector axioms only in cases when their absence would result in unsound results.

One of the main challenges related to user-propagation in general is that several SMT solving optimizations are not compatible with our lazy encoding approach. For example, pure literal elimination cannot be applied lazily, as it is unknown whether a literal is actually pure or not. Another line of further work comes with applying user-propagators in SMT over quantified

formulas, in order to improve model-based quantifier instantiation, for example in combination with functions over booleans or bit-vectors. Supporting further theories directly in the user-propagator, like floating-points or algebraic datatypes, would a further extension that may be worth considering.

We finally note, that although we mainly discussed lazy clause generation and focused on investigated how custom boolean-based theories can be utilized, more complex non-finite theories can be modeled as well in using user-propagators on top of SMT solving, by observing (dis-)equality constraints and checking the consistency of the theory-specific atoms in the fixed and final callbacks. However, further callbacks may be required to efficiently implement such more complex theories.

Acknowledgments

We thank Nuno Lopes (U. Lisbon) for valuable discussion on potential applications of user-propagators in alive2 benchmarks. The work described in this extended abstract was partially supported by the ERC Consolidator Grant ARTIST 101002685 and the Austrian FWF project W1255-N23.

References

[1] D. Jovanovic, L. de Moura, Solving Non-Linear Arithmetic, ACM Commun. Comput. Algebra 46 (2012) 104–105. doi:10.1145/2429135.2429155.

[2] M. Berzish, V. Ganesh, Y. Zheng, Z3str3: A String Solver with Theory-aware Heuristics, in: FMCAD, 2017, pp. 55–59. doi:10.23919/FMCAD.2017.8102241.

[3] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, X. Zhang, Z3str2: an Efficient Solver for Strings, Regular Expressions, and Length Constraints, Formal Methods Syst. Des. 50 (2017) 249–288. doi:10.1007/s10703-016-0263-6.

[4] A. Reynolds, A. Nötzli, C. W. Barrett, C. Tinelli, A Decision Procedure for String to Code Point Conversion, in: IJCAR, 2020, pp. 218–237. doi:10.1007/978-3-030-51074-9_13.

[5] S. Kan, A. W. Lin, P. Rümmer, M. Schrader, CertiStr: a Certified String Solver, in: CPP, 2022, pp. 210–224. doi:10.1145/3497775.3503691.

[6] L. Kovács, S. Robillard, A. Voronkov, Coming to Terms with Quantified Reasoning, in: POPL, 2017, pp. 260–270. doi:10.1145/3009837.3009887.

[7] A. Reynolds, V. Kuncak, Induction for SMT Solvers, in: VMCAI, 2015, pp. 80–98. doi:10.1007/978-3-662-46081-8_5.

[8] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. W. Barrett, C. Tinelli, Towards Satisfiability Modulo Parametric Bit-vectors, J. Autom. Reason. 65 (2021) 1001–1025. doi:10.1007/s10817-021-09598-9.

[9] A. Niemetz, M. Preiner, C. Wolf, A. Biere, Btor2 , BtorMC and Boolector 3.0, in: CAV, 2018, pp. 587–595. doi:10.1007/978-3-319-96145-3_32.

[10] L. M. ça de Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: TACAS, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.

[11] O. Ohrimenko, P. J. Stuckey, M. Codish, Propagation via Lazy Clause Generation, Constraints An Int. J. 14 (2009) 357–391. doi:10.1007/s10601-008-9064-x.
[12] T. Feydy, P. J. Stuckey, Lazy Clause Generation Reengineered, in: CP, 2009, pp. 352–366. doi:10.1007/978-3-642-04244-7_29.
[13] E. Giunchiglia, Y. Lierler, M. Maratea, Answer Set Programming Based on Propositional Satisfiability, J. Autom. Reason. 36 (2006) 345–377. doi:10.1007/s10817-006-9033-2.
[14] R. Kaminski, T. Schaub, P. Wanko, A Tutorial on Hybrid Answer Set Solving with clingo, in: Proc. of Reasoning Web, 2017, pp. 167–203. doi:10.1007/978-3-319-61033-7_6.
[15] N. Bjørner, Engineering theories with Z3, in: Proc. of APLAS, 2011, pp. 4–16. doi:10.1007/978-3-642-25318-8_3.
[16] Y. Zheng, X. Zhang, V. Ganesh, Z3-str: a Z3-Based String Solver for Web Application Analysis, in: FSE, 2013, pp. 114–124. doi:10.1145/2491411.2491456.
[17] R. Bruttomesso, E. Pek, N. Sharygina, A. Tsitovich, The OpenSmt Solver, in: TACAS, 2010, pp. 150–153. doi:10.1007/978-3-642-12002-2_12.
[18] N. Bjørner, M. Levatich, N. P. Lopes, A. Rybalchenko, C. Vuppalapati, Supercharging Plant Configurations Using Z3, in: P. J. Stuckey (Ed.), CPAIOR, 2021, pp. 1–25. doi:10.1007/978-3-030-78230-6_1.
[19] J. Lee, C. Hur, R. Jung, Z. Liu, J. Regehr, N. P. Lopes, Reconciling High-Level Optimizations and Low-Level Code in LLVM, in: OOPSLA, 2018, pp. 125:1–125:28. doi:10.1145/3276495.