Using Incremental SMT Solving from a Datalog-based System

Aaron Bembenek
Harvard University
Cambridge, MA USA

Michael Ballantyne
Northeastern University
Boston, MA USA

Michael Greenberg
Pomona College
Claremont, CA USA

Nada Amin
Harvard University
Cambridge, MA USA

1 Introduction

Satisfiability modulo theories (SMT) solving is a powerful tool. However, any client of an SMT solver faces the challenge of using the solver efficiently. In particular, modern SMT solvers support incremental solving, so there is often an advantage to asking queries in an order that lets the solver reuse work from earlier queries when answering new ones. Thus, clients are incentivized to pose queries in a way that has good "solver locality" and is amenable to effective incremental solving.

Because they are both used for automated reasoning tasks, there is a natural appeal to augmenting logic programming systems with the capabilities of SMT solving. One such hybrid system, Formulog [3], does this by extending Datalog with terms that represent SMT formulas and an interface to an external SMT solver. However, the notion of "solver locality" raises a potential problem for Formulog and other logic programming systems that embrace Kowalski's principle [8] of separating the logic of a computation from its control: The programmer cannot easily control the order of computations, and the language runtime might perform them in an order that has low solver locality.

For example, consider using Formulog to explore simple paths in a directed graph whose edges are labeled with SMT propositions. A path is explored only if the conjunction of its edge labels is satisfiable. This computation would have clearly good solver locality if it consecutively checked the satisfiability of similar paths (e.g., paths with a large common prefix); such a result might be achieved by using a sequential depth-first search (DFS) of the graph, starting at each node in turn. However, the same is not obviously the case for Formulog: because of its parallelized bottom-up evaluation algorithm, it effectively performs a breadth-first search of the graph starting from each node of the graph in parallel. Thus, paths discovered around the same time might not share a very large common prefix; in fact, they might be completely disjoint! Given the lack of obvious solver locality, it is not clear if this Formulog program should expect any advantage from incremental SMT solving.

We show empirically that, even in this setting, incremental SMT solving can in fact speed up logic programs that make SMT queries. We do so by evaluating the effectiveness of two lightweight encoding mechanisms that mediate between Formulog and an external SMT solver. The mechanisms use different capabilities defined in the SMT-LIB standard [2]: the first explicitly manages a solver's assertion stack, while the second checks satisfiability under retractable assumptions. The latter strategy consistently leads to speedups over a non-incremental baseline across a variety of benchmarks involving different SMT solvers and SMT-LIB logics, suggesting that logic programming systems can indeed take advantage of incremental, black-box SMT solving.

2 Making incremental SMT calls

A Formulog user builds complex terms representing SMT formulas and reasons about them with built-in operators like `is_sat` and `get_model`. When an SMT operator is invoked, the Formulog runtime takes

To appear in EPTCS.

the SMT query, serializes it into the SMT-LIB format, ships it off to an external solver, and returns the result. The serialization code takes a list of conjuncts as input; its job is to assert these conjuncts to the SMT solver in a way that leads to efficient solving. We evaluate three different strategies for SMT clients: a baseline non-incremental strategy and two incremental strategies.

First, some background: SMT-LIB-compatible solvers maintain a stack of assertion frames. Frames can be pushed on and popped off the stack using the commands `push` and `pop`. When a frame is popped, the solver's state is reset to the state it had before that frame was added, forgetting any assertions made since the corresponding `push` command along with any consequences derived from those assertions.

Our baseline, non-incremental strategy clears the solver's assertion state between every satisfiability check: When it serializes an SMT query, it generates code that pushes a fresh frame on the stack, asserts each conjunct, checks for satisfiability, and then pops the frame it added.

Our second strategy (PP) explicitly `push`es and `pop`s the solver's stack to enable incremental solving. When serializing a query, PP `pop`s off frames until the solver's assertion stack is a prefix of the current list of conjuncts, `push`es assertions for each of the remaining conjuncts of the query, and then checks for satisfiability. PP provides incremental solving when adjacent queries share large common prefixes. PP works well for clients who explore a constraint space using DFS (since the stack disciplines match up), but penalizes clients who use search techniques that align less well with using a stack (breadth-first search, heuristic searches, etc.).

Our final strategy (CSA) uses the `check-sat-assuming` SMT-LIB command, which checks the satisfiability of the solver's assertion state assuming particular truth values for some given boolean variables. CSA uses a layer of indirection to treat the solver's assertion state as a cache of assertions that can be enabled or disabled for a particular query [6]: instead of asserting a conjunct ϕ directly, CSA asserts the implication $x \implies \phi$, where x is a fresh boolean variable. To check the satisfiability of a query including ϕ, we include x in the list of literals provided to the `check-sat-assuming` command. Formulog's CSA client maintains a map from conjuncts to boolean variables to keep track of which variables should be enabled for a particular query and to ensure that no conjunct is ever asserted more than once.

SMT clients using CSA need not manage an explicit stack of assertions; there is no penalty for exploring the constraint space in a way that does not align well with using a stack. Intuitively, CSA should be a boon for logic programming systems like Formulog that do not use a DFS-based search.

3 Results

We empirically evaluated two hypotheses: **(H1) Formulog should be able to take advantage of incremental SMT solving**; i.e., either PP or CSA should outperform the non-incremental baseline strategy on most benchmarks; **(H2) CSA should deliver more consistent and more substantial speedups than PP**. To evaluate these hypotheses, we tested the relative performance of the three strategies on three Formulog programs given a variety of inputs.

The first program performs symbolic execution [7] over a simple imperative language. We ran the tool on six different input programs. We tried the solvers Z3 (v4.8.8) [9], CVC4 (v1.7) [1], and Boolector (v3.2.1) [10] (all set to use the logic QF_ABV). In all 18 cases, the baseline was bested by an incremental strategy (H1); CSA achieved the best speedup in 12 of these cases (H2).

The second program type checks programs written in Dminor [4], a language with refinement types that are encoded as SMT formulas. We ran the type checker on three input programs, using the solver Z3 with the logic ALL. CSA had the best improvement over the baseline in all three cases (H1, H2).

The third program computes reachability on graphs with edges labeled by propositions; a path is fea-

sible only if the conjunction of the propositions along that path is satisfiable. We computed reachability on 24 random graphs in various bit vector and linear integer arithmetic (LIA) logics. We used the solvers Z3, CVC4, and Yices (v2.6.2) [5], and additionally Boolector for the graphs with bit vector formulas. The incremental strategies were better than the baseline on 60 of 84 experiments (H1), and CSA had the best improvement in 59 cases (H2). Of the cases where there was no gain by using incremental solving, 14 were clustered in the experiments with logics combining LIA and arrays.

Overall, out of 105 experiments, either PP or CSA beat the non-incremental baseline in 81 cases (**H1✓**). CSA provided a speedup in 79 cases, compared to 39 for PP; there were only two cases where PP provided a speedup and CSA did not, and only five where both strategies led to speedups but PP was faster than CSA (**H2✓**). Thus, our evaluation confirms our hypotheses: CSA is an effective, lightweight mechanism for integrating a logic programming system with an incremental SMT solver.

4 Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-19-C-0004. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA). We thank Arlen Cox for his perspective on incremental SMT solving and his pointers on the origins of checking satisfiability with assumptions. William E. Byrd also provided valuable feedback and guidance.

References

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In: *Proc. 23rd Int. Conf. on Computer Aided Verification*, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[2] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`.

[3] Aaron Bembenek, Michael Greenberg & Stephen Chong (2020): *Formulog: Datalog for SMT-based Static Analysis*. In submission.

[4] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu & David Langworthy (2012): *Semantic Subtyping with an SMT Solver*. *Journal of Functional Programming* 22(1), pp. 31–105, doi:10.1017/S0956796812000032.

[5] Bruno Dutertre (2014): *Yices 2.2*. In: *Proc. 26th Int. Conf. on Computer Aided Verification*, pp. 737–744, doi:10.1007/978-3-319-08867-9_49.

[6] Niklas Eén & Niklas Sörensson (2003): *Temporal induction by incremental SAT solving*. *Electronic Notes in Theoretical Computer Science* 89(4), pp. 543–560, doi:10.1016/S1571-0661(05)82542-3.

[7] James C. King (1976): *Symbolic Execution and Program Testing*. *Commun. ACM* 19(7), pp. 385–394, doi:10.1145/360248.360252.

[8] Robert Kowalski (1979): *Algorithm = Logic + Control*. *Commun. ACM* 22(7), pp. 424–436, doi:10.1145/359131.359136.

[9] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proc. 14th Int. Conf. on Tools and Algs. for the Construction and Analysis of Systems*, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[10] Aina Niemetz, Mathias Preiner & Armin Biere (2014 (published 2015)): *Boolector 2.0 System Description*. *Journal on Satisfiability, Boolean Modeling and Computation* 9, pp. 53–58, doi:10.3233/SAT190101.