Making Incremental SMT Solving Work for Logic Programming Systems

Aaron Bembenek[1], Michael Ballantyne[2],
Michael Greenberg[3], Nada Amin[1]
[1]*Harvard University,* [2]*Northeastern University,* [3]*Pomona College*

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

There is a natural appeal to connecting logic programming systems to satisfiability modulo theories (SMT) solvers, as both have proven to be useful tools for automated reasoning tasks. However, a practical challenge is how to do this while taking advantage of incremental SMT solving, a key performance consideration. In other settings, it is possible for the programmer to structure computation so that SMT calls are made in an order with good "solver locality," and thus amenable to effective incremental solving. This approach does not work for many logic programming languages where, by design, the programmer does not have as much control over the order of evaluation. This paper explores whether, and how, such systems can still take advantage of incremental SMT solving. In particular, it empirically evaluates the effectiveness of two lightweight encoding mechanisms that mediate between Formulog (a Datalog-like language) and an external SMT solver. The mechanisms use different capabilities defined in the SMT-LIB standard: the first explicitly manages a solver's assertion stack, while the second checks satisfiability under retractable assumptions. We show that 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.

KEYWORDS: SMT, Datalog, Formulog

1 Introduction

SMT solving is a powerful tool for automated reasoning, with applications in domains such as model checking (Cimatti and Griggio 2012), program synthesis (Jha et al. 2010), and program verification (Leino 2010). Given logic programming's historical application to automated reasoning tasks, there is a natural appeal to augmenting logic programming systems with the capabilities of SMT solving. One such hybrid system is Formulog (Bembenek et al. 2020), a Datalog-like language with terms that represent SMT formulas and an interface to an external SMT solver.

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 such that the solver can use 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.

The notion of "solver locality" raises a potential problem for logic programming systems that embrace Kowalski's principle of separating the logic of a computation from its control (1979). In other settings, the programmer can explicitly structure the order of a computation so that queries have good solver locality. In logic programming, however, the programmer cedes control to the runtime. For example, in the case of Formulog, the runtime might aggressively rewrite programs, and then evaluate them using a parallelized bottom-up saturation algorithm. It is hard for the programmer to craft computations with good solver locality in such a setting.

Indeed, it is not easy to say *a priori* if there is any gain to be had for a system like Formulog in incremental SMT solving. Take, for example, the case of using Formulog to compute all-pairs reachability of a directed graph whose edges are labeled with SMT propositions. Further, say that reachability is modulo path satisfiability — a path is considered feasible only if the conjunction of its edge propositions is satisfiable — and that the computation eagerly prunes infeasible paths by checking newly discovered paths with the SMT solver. This computation could 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, which — thanks to its parallelized bottom-up evaluation algorithm — 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 incremental SMT solving can in fact speed up logic programs that make SMT queries. Furthermore, this gain can be achieved through lightweight mechanisms sitting between the logic programming runtime and the external SMT solver; in the spirit of Kowalski's principle, these mechanisms are invisible to the programmer.

Specifically, this paper evaluates the effectiveness of two different strategies a logic programming runtime can use when it poses SMT problems to an external solver that implements the SMT-LIB standard (Barrett et al. 2017). The first strategy explicitly manages the external solver's assertion stack using `push` and `pop` commands; these commands are the traditional mechanism for incremental SMT solving, but the stack discipline works best when the problem space is itself explored using a DFS — i.e., a stack discipline. The second strategy takes advantage of checking satisfiability under retractable assumptions using the `check-sat-assuming` command. Unlike `push` and `pop`, the `check-sat-assuming` approach does not have a pro-DFS bias — intuitively, a better fit for systems like Formulog that do not use DFS-based evaluation algorithms.

Our experimental results confirm this intuition: Using Formulog as our guinea pig system, we show that the strategy based on `check-sat-assuming` provides some degree of speedup over a non-incremental baseline on 79 of 105 benchmarks (75%), whereas the `push`/`pop` based strategy provides a speedup on only 39 benchmarks (37%). Of the 37 benchmarks where both strategies provide a speedup, the approach based on `check-sat-assuming` provides a larger speedup in 32 cases. Our evaluation uses a range of SMT solvers and SMT logics, giving us confidence that our results generalize and accurately advertise the potential advantages of black-box, incremental SMT solving for logic programming systems.

We claim as contributions:

- A characterization of three possible strategies for communicating with an SMT solver (Section 2) from a logic programming language (Section 3).
- An evaluation of these three strategies on a range of tasks: symbolic execution, refinement type checking, and proposition graph reachability (Section 4). We offer evidence that the `check-sat-assuming` strategy is generally the most effective approach.

Section 5 addresses the limitations of our approach, and discusses related work.

2 Framing SMT problems

Any application using an SMT solver has to make several important decisions. One decision is whether to run the solver in single-shot mode, where each query is treated independently, or in incremental mode, where the solver can retain information from query to query. Incremental mode often — but not always — results in speedups: in addition to sharing query data structures, the solver can remember things it learned about the shared parts. For the rest of this paper, we take for granted that the solver is running in incremental mode.

A second decision — the one at the heart of this paper — is how to frame SMT problems in a way that leads to effective incremental solving. Faced with a conjunction of SMT propositions, there are many different ways of posing them to an SMT solver to check satisfiability. Just because the solver is in incremental mode does not mean that a client can blithely assert conjuncts and expect a gain from incremental solving; rather, care needs to be taken to formulate queries in a way such that the solver generates intermediate results that can be reused when evaluating later queries. Furthermore, a given strategy might lead to good incremental solving performance for some solver workloads and slowdowns for other workloads.

This section describes the strategies we evaluate in this work. We start with our baseline strategy, with no incremental solving, and then continue on to two incremental strategies (`push`/`pop` and `check-sat-assuming`). Throughout, we use the running example of asking two similar queries over the boolean variables x, y, and z:

$$\begin{gathered}(x \lor y) \land (\neg x \lor \neg z) \land \quad y \land z \\ (x \lor y) \land (\neg x \lor \neg z) \land \neg y \land z\end{gathered}$$

The queries differ in their third conjunct concerning y; the first query is satisfiable, but the second one is not — the first two conjuncts imply that $y \lor \neg z$ holds.

2.1 The baseline strategy

Our baseline strategy does not take advantage of incremental solving at all: Between every satisfiability check, we clear the solver's assertion state. To make an SMT query, we first `push` a new frame on the solver's assertion stack, then we make our assertions and check for satisfiability, and finally we pop off this new frame with the command `pop`. At the end, the solver is back in the state it was in before the initial `push` command. On our example, this strategy gives us:[1]

```
(push) (assert (or x y))
(assert (or (not x) (not z)))
(assert y)
(assert z)
(check-sat) (pop)
(push) (assert (or x y))
(assert (or (not x) (not z)))
(assert (not y))
(assert z)
(check-sat) (pop)
```

[1] We use SMT-LIB's s-expression syntax; we omit variable declarations and use `push` for `push 1` and `pop` for `pop 1`.

There is no sharing between calls; we repeat assertions and lose state. For example, after processing the first query, the solver will forget that $y \vee \neg z$ holds, even though it is derivable from conjuncts common to both queries. While this strategy does not use incremental solving, it still uses `push` and `pop` and the solver must be in incremental solving mode.

2.2 The `push`/`pop` strategy

Incremental solving with an SMT solver is traditionally done by explicitly managing a stack of assertion frames using the commands `push` and `pop`. We call this strategy the `push`/`pop` or PP strategy. The command `push` pushes an assertion frame on the stack, and the command `(pop N)` pops `N` frames off the stack. When an assertion frame is popped off the stack, the solver's state is reset to the state it had before that frame was added, forgetting any assertions made since the last `push` along with any consequences derived from those assertions.

When posing a query using the PP strategy, we begin by popping off as many frames as necessary so that the solver's assertion stack is a prefix of the current query. We then assert the remaining conjuncts of the query, placing a `push` command before every assertion. We would encode our example as:

```
(push) (assert (or x y))
(push) (assert (or (not x) (not z)))
(push) (assert y)
(push) (assert z)
(check-sat)
(pop 2)
(push) (assert (not y))
(push) (assert z)
(check-sat)
```

The PP strategy takes advantage of *some* sharing between queries; for example, it remembers that $y \vee \neg z$ must hold, since the first two conjuncts remain the same. However, it will forget that z must be true, as this conjunct is popped off the stack (only to be reasserted again). In general, the PP strategy works well for clients who explore a constraint space using depth-first search (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.). We can see this mismatch in microcosm here: the fourth conjunct z must be asserted twice given the current ordering of conjuncts. If the third and fourth conjunct were swapped, then z would only be asserted once, and it would not be forgotten between calls.

2.3 The `check-sat-assuming` strategy

Our third and final strategy takes advantage of an alternative approach to incremental solving: SMT solvers can check for the satisfiability of a set of assertions while assuming particular truth values for some boolean variables. This technique was developed in the context of incremental SAT solving (Eén and Sörensson 2003) and added to the SMT-LIB standard in version 2.5 in 2015. We call this the `check-sat-assuming` or CSA strategy.

The `check-sat-assuming` command takes a list of a boolean variable and checks for the satisfiability of the current assertions, assuming the boolean variables in the list.[2] As a client no longer needs to manage an explicit stack of assertions, the client is not penalized for exploring the search space in a way that does not align well with using a stack. Intuitively, this should be a boon for logic programming systems like Formulog that do not use a DFS-based search.

In the CSA strategy, assertions are made under a level of indirection: instead of asserting a conjunct ϕ directly, we assert $a \implies \phi$, where a is a fresh boolean variable; we refer to these variables as "assumption variables." When we want to check the satisfiability of a query including ϕ, we include a in the list of literals provided to the `check-sat-assuming` command. On our example queries, this gives us:

```
(assert (=> a0 (or x y)))
(assert (=> a1 (or (not x) (not z))))
(assert (=> a2 y))
(assert (=> a3 z))
(check-sat-assuming (a0 a1 a2 a3))
(assert (=> a4 (not y)))
(check-sat-assuming (a0 a1 a4 a3))
```

Each variable aN is an assumption variable. The CSA strategy allows a high degree of sharing between satisfiability checks: the fact that three of the four conjuncts are the same in each query is directly reflected in the fact that the lists of literals supplied to each `check-sat-assuming` command includes the assumption variables `a0`, `a1`, and `a3`. The CSA strategy allows for more sharing than in the PP strategy, where only two assertions are remembered between calls.

2.4 Handling `unknown`

In general, an SMT solver returns one of three results for a satisfiability check: `sat` for satisfiable, `unsat` for unsatisfiable, and `unknown` when the solver is not able to determine an answer. This third case arises when working in logics that are undecidable (for example, non-linear integer arithmetic, or various fragments with universal quantifiers). Whether a solver returns `unknown` to a query depends partly on how that query is encoded. Unfortunately, the three strategies above cannot be used interchangeably. In particular, the CSA strategy seems to result in `unknown` for queries that the other strategies do not. To rule out these "spurious" `unknown`s, it is necessary for the CSA strategy to double-check an `unknown` result using one of the other strategies (in our experiments, it uses the PP strategy). Thus, the CSA strategy incurs some additional overhead when working with theories that might yield `unknown`.

3 System design

A Formulog user builds complex terms representing SMT formulas and reasons about them with built-in operators like `is_sat` and `get_model`. The `is_sat` operator takes a list of formulas, translates them to SMT assertions, and queries an external solver to determine whether their conjunction is satisfiable. It returns one of three values: `sat`, `unsat`, or `unknown`.[3] Formulog

[2] Technically, it accepts a list of literals, where a literal is a boolean variable or its negation. In our experiments, we only include positive literals, as we found including negative literals led to worse performance.
[3] Our description of the Formulog SMT interface is slightly simplified, but captures the key parts relevant to this paper.

```
r(X, Y, Z) :- p(X), q(Y, Z),
              Phi1 = ‘X #= bv_add(Y, Z)’,
              Phi2 = ‘Y #= bv_mul(2, Z)’,
              is_sat([Phi1, Phi2]) = unsat.
```

Fig. 1. Example Formulog program.

```
for X in p:
  for (Y, Z) in q:
    Phi1 = ‘X #= bv_add(Y, Z)’
    Phi2 = ‘Y #= bv_mul(2, Z)’
    if query_solver([Phi1, Phi2]) == unsat: r += (X, Y, Z)
```

Fig. 2. Evaluation of the example Formulog program in Figure 1.

provides a library of constructors for building formulas involving SMT-LIB constructs such as uninterpreted functions, integers, bit vectors, arrays, and algebraic data types.

For a concrete example, consider the Formulog rule in Figure 1. The body of this rule first looks up a value x in relation `p` and a pair (y,z) in relation `q`. It then assigns to `Phi1` and `Phi2` terms representing the formulas $x = y + z$ and $y = 2 * z$, respectively, where x, y, and z are interpreted as bit vectors; formula terms are single-quoted. If the conjunction of these formulas is unsatisfiable, the rule succeeds and derives the tuple (x,y,z) for the `r` relation. Formulog uses a bottom-up saturation strategy in the style of semi-naive evaluation (Bancilhon 1986). Conceptually, our example rule is evaluated like the pseudocode in Figure 2. Relation lookups translate to nested `for` loops, and the operator `is_sat` translates to a call to the function `query_solver`. The `query_solver` function mediates the interaction between Formulog evaluation and SMT solving: It encodes the given list of assertions into SMT-LIB, transmits the encoded assertions to an external solver, and relays back the result.

Different versions of the function `query_solver` implement the three strategies we explore in this paper: namely, the baseline non-incremental strategy, the PP strategy, and the CSA strategy. In each case, the function treats each member of its input list as an independent assertion.

The baseline version of `query_solver` maintains no state; upon receiving assertions, it issues a `push` command, poses the assertions, checks for satisfiability, and then issues a `pop` command.

The PP version of `query_solver` maintains a stack of assertions, with the invariant that the solver currently has an equivalent stack of assertions. When `query_solver` is invoked with a list of assertions, it finds the largest shared prefix of its stack and the list, starting with the bottom of the stack and the *end* of the list, and moving upwards through the stack and backwards through the list.[4] Stack items beyond this prefix are popped from the solver; list items beyond this prefix are added to the solver (in the reversed iteration order), each one preceded by a `push` command.

The CSA version of `query_solver` maintains a map from assertions to assumption variables, with the invariant that an entry $\phi \mapsto a$ in the map means that the external solver has the assertion

[4] Linked lists in Formulog are “consed” together as stacks: i.e., a new head element is added to the front of a preexisting list, so the last element of the list represents the longest-standing member of the list, in the same way that the bottom-most element is the longest-standing member of the stack.

$a \implies \phi$ in its assertion stack. When we first meet a new assertion ϕ, a fresh solver variable a is created, the map is updated with an entry from this assertion to the variable, and the implication $a \implies \phi$ is added to the solver. Once all assertions have been accounted for, a call is made to CSA with the relevant assumption variables enabled.

Whether or not these strategies provide good solver locality depends on how a given Formulog program generates SMT queries. For instance, consider evaluating the example program in Figure 1 on these (ordered) relations:

```
p = [a, b]; q = [(c, d), (e, f)]
```

where a, b, etc. refer to arbitrary terms of an appropriate bit vector type. Doing so will result in the `query_solver` function being invoked with four lists, in this order:

```
[‘a #= bv_add(c, d)’, ‘c #= bv_mul(2, d)’]
[‘a #= bv_add(e, f)’, ‘e #= bv_mul(2, f)’]
[‘b #= bv_add(c, d)’, ‘c #= bv_mul(2, d)’]
[‘b #= bv_add(e, f)’, ‘e #= bv_mul(2, f)’]
```

There are some shared assertions between these lists — in particular, `‘c #= bv_mul(2, d)’` and `‘e #= bv_mul(2, f)’` each occur in two lists — but there will be no sharing between any of the solver calls using the `push-pop` approach, as no two consecutive lists (read backwards) share a non-empty prefix. On the other hand, the `check-sat-assuming` approach will cache and reuse these shared assertions.

As another wrinkle, the `for` loops in Figure 2 are parallelized in Formulog: different iterations of the loops happen in different threads. Each evaluation thread is associated with its own solver instance, and the stateful versions of the function `query_solver` maintain thread-local state that correctly reflects the state of the corresponding solver for that thread. The distribution of queries between threads has a direct impact on the extent to which the different versions of `query_solver` can share assertions. In this example, if the first and third queries wind up on the same thread and the second and fourth queries on another, then both the PP and CSA versions will get sharing between calls. On the other hand, if the first two queries are on one thread and the second two on another, then neither strategy will have any sharing.

The bottom-up evaluation strategy, and the parallelism it enables, are important design elements of Formulog, which is targeted for building scalable SMT-based static analyses. However, these design choices make it hard to predict how effectively the different `query_solver` versions will share formulas during incremental solving.

4 Evaluation

This section empirically evaluates two hypotheses. **First, it should be possible for Formulog to take advantage of incremental SMT solving.** We consider this hypothesis to hold if, on many benchmarks, either the PP strategy or the CSA strategy outperforms the non-incremental baseline strategy. **Second, given our context, the CSA strategy should deliver more consistent and more substantial speedups than the PP strategy**.

The evaluation is in three case studies. The first two are substantial static analyses written in Formulog: a symbolic execution tool (Section 4.1) and a refinement type checker (the only case study to induce `unknown` results; Section 4.2). The third case study computes reachability on graphs with edges labeled by propositions (Section 4.3); a path is feasible only if the conjunction of the propositions along that path is satisfiable. Section 4.4 summarizes our results, confirming that our two hypotheses do in fact hold.

For all experiments, we used an Ubuntu Server 16.04 LTS machine with a 3.1 GHz Intel Xeon Platinum 8175 processor (24 physical CPUs/48 virtual CPUs) and 192 GiB of memory. We set the Formulog runtime to use 40 threads. For each result, we report the median of three trials. Instead of absolute times, we report times relative to the non-incremental baseline strategy; we calculate this by dividing the baseline time by the time for the encoding strategy in question.[5] Thus, speedups correspond to numbers greater than 1, and slowdowns to numbers less than 1. In cases where a strategy times out, we treat its time as being the timeout limit for that case study. In tables, we identify slowdowns relative to the baseline in red; we identify the best speedup for a benchmark with **bold** font. We mark with a dagger † cases in which some trials timed out and with an asterisk * cases in which some trials crashed.

In the experiments, we variously use the solvers Z3 v4.8.8 (de Moura and Bjørner 2008), CVC4 v1.7 (Barrett et al. 2011), Yices v2.6.2 (Dutertre 2014), and Boolector v3.2.1 (Niemetz et al. 2015). The first three support a wider range of SMT-LIB theories than Boolector, which is limited to bit vectors, arrays, and uninterpreted functions.

4.1 Case study: symbolic execution

Our first case study is a symbolic execution tool for a subset of LLVM bitcode (Lattner and Adve 2004) that corresponds to a simple imperative language with arrays of machine integers. Some of these machine integers can be symbolic, which means they represent unknown values. Symbolic execution is a program testing technique for evaluating programs with symbolic values (King 1976; Cadar and Sen 2013). Initially, a symbolic value is assumed to be any value in its domain. As the interpreter progresses through the program, we narrow the possible values each symbolic value can have. When the interpreter reaches a branch, it forks and takes both branches, constraining the state in each fork to be consistent with the branch that has been taken. Each interpreter branch might directly or indirectly constrain the possible values a symbolic value can take on. Our symbolic execution tool represents symbolic machine integers as bit vector-valued SMT formulas, and maintains a state that constrains these values via SMT propositions. Whenever a branch is taken that is conditioned on a symbolic value, we invoke the SMT solver to make sure that branch is actually feasible — if not, we can "prune" the branch and ignore it. Our symbolic executor is $\sim$800 lines of Formulog.

We have evaluated the symbolic evaluator on six different programs, drawn from three template programs. The first template (shuffle-*N*) shuffles an array of *N* symbolic integers and asserts that the resulting array represents the same set of inputs as the initial array. The second template (sort-*N*) uses selection sort to sort an array of *N* symbolic integers and asserts that the resulting array is sorted. The third template (numbrix) completes a partially filled-in 4$\times$4 grid with the integers 1 through 16, such that there is a path through the grid from 1 to 16 following consecutive integers; the path uses only vertical and horizontal movements. We evaluate the path-finding template on both satisfiable (numbrix-sat) and unsatisfiable (numbrix-unsat) grid instances. These templates lead to very different symbolic execution behavior, both in terms of number of program paths explored (which effects how Formulog's parallelism behaves, since paths are evaluated in parallel) and the number of SMT calls made.

[5] Tables with absolute times can be found in Appendix A.

Table 1. Speedups for each strategy in each solver on symbolic execution benchmarks

			Z3		CVC4		Boolector	
Benchmark	# paths	# SMT calls	PP	CSA	PP	CSA	PP	CSA
shuffle-4	125	455	**1.58**	**1.58**	**5.46**	4.68	15.94	**16.67**
shuffle-5	1,296	4,088	2.83	**3.16**	**13.33**	10.78	1.26	**2.93**
sort-6	1,359	18,798	0.90	**1.93**	1.00†	**60.00**	1.17	**11.89**
sort-7	11,035	178,461	0.90	**1.71**	1.00†	**7.69**	1.00†	**7.73**
numbrix-sat	1	97	**1.51**	1.11	**1.07**	0.91	**1.69**	**1.69**
numbrix-unsat	1	97	**1.47**	1.11	**1.31**	1.09	1.50	**1.55**
Average			1.53	**1.77**	3.86	**14.19**	3.76	**7.08**
Median			1.49	**1.65**	1.19	**6.18**	1.38	**5.33**

We ran experiments using the SMT solvers Z3, CVC4, and Boolector. We set each solver to use the logic `QF_ABV` (quantifier-free arrays and bit vectors), although the sort-N benchmarks do not actually generate formulas with array constructs. Timeouts were set to 30 minutes.

Incremental solving led to some form of speedup for every benchmark and solver (Table 1). Results were mixed in terms of which strategy was better. On the shuffle-N benchmarks, the CSA strategy was generally better for Z3 and Boolector, whereas the PP strategy was better for CVC4. The CSA strategy was the unambiguous champion on the sort-N benchmarks, where the PP strategy timed out on half of the experiments and also had slowdowns relative to the baseline. This result intuitively makes sense, as the sort-N benchmarks have the symbolic executor explore a large number of paths using, effectively, a breadth-first search—exactly the opposite of what the PP strategy needs On the other hand, the PP strategy was in general more effective for the numbrix benchmarks. Each of these benchmarks have only a single path for the symbolic executor to explore, making them a good fit for the PP strategy, and making the additional flexibility of the CSA strategy unnecessary.

4.2 Case study: refinement type checking

Our second case study is a type checker for Dminor, a first-order functional programming language with refinement types (Bierman et al. 2012). Dminor employs semantic subtyping: a type is a subtype of another if its logical denotation implies the denotation of the supertype. The type checker uses SMT solving primarily to prove subtyping relations, although it also uses it to prove that an expression is pure (i.e., is deterministic and terminates) and to prove that a type is empty (i.e., its denotation implies false). The encoding of types as logical formulas is complex, requiring SMT-LIB features including arrays, algebraic data types, uninterpreted functions, uninterpreted sorts, and universally quantified axioms. Our type checker is $\sim$1.2K lines of Formulog.

We have evaluated the type checker using the three SMT strategies on three benchmarks. To the best of our knowledge, only nine Dminor programs are open source; of these, the Formulog-based type checker can handle six (the others require the ability to generate an instance of a type, an ancillary feature not supported by the Formulog-based type checker). The most substantial of these programs is an interpreter for a simple imperative language. We have combined these programs together to form a composite program of $\sim$150 LOC. For each benchmark, we further

Table 2. Speedups for each strategy on refinement type checking benchmarks

		PP		CSA	
Benchmark	# SMT calls	Speedup	# `unknown`	Speedup	# `unknown`
all-1	1,006	**1.91**	4	**1.91**	4
all-10	9,619	10.99	40	**13.86**	40
all-100	89,597	12.04	400	**17.82**	546
Average		8.31		**11.20**	
Median		10.99		**13.86**	

compose this program on top of itself: the benchmark all-N consists of N copies of the program. We tested the configurations all-1, all-10, and all-100. Z3 was the only back-end SMT solver we could use (the other solvers we considered did not have sufficient support for algebraic data types). Timeouts were set to one hour.

Incremental solving led to consistent speedups on these benchmarks, with the CSA strategy being the clear winner (Table 2). For the largest case (all-100), the CSA strategy has a speedup of 17.82× over the baseline strategy, compared to a speedup of 12.04× for the PP strategy. These speedups might be even more dramatic than reported, as the baseline timed out but we conservatively treat it as completing in an hour. Because the formulas generated by the type checker include universal quantifiers, the SMT solver sometimes returns `unknown`. As mentioned in Section 2, the CSA strategy sometimes returns a "spurious" `unknown` when the PP strategy does not. In all-100, 146 of the 546 `unknowns` returned by the CSA strategy were spurious. Although this is a large fragment of the `unknowns` (27%), only a small percentage (0.6%) of calls overall result in `unknown`. The number of spurious `unknowns` seems to be correlated with the size of the benchmark (and presumably with the complexity of the solver state), as all-1 and all-10 had none of them, and all-50 (not reported in the table) had a median of only one.

4.3 Case study: proposition graph reachability

In this case study, we compute all-pairs reachability for proposition graphs, i.e., directed graphs where edges are labeled with SMT propositions. Reachability is modulo the satisfiability of the propositions along a path. The program in this case study consists of two rules that compute tuples of the `reach` relation, where $\texttt{reach}(x, y, \vec{z}, \vec{\phi})$ holds if there is a path $\vec{z}$ from node x to node y with the satisfiable path constraint $\vec{\phi}$. Because it is easy to synthesize random proposition graphs with different types of SMT propositions, this case study gives us the opportunity to evaluate how well our hypotheses generalize across different SMT-LIB logics — a handy thing to know, since SMT solvers often use internal solvers specialized to particular logics.

We generate random proposition graphs in a given theory in two phases: first, we generate a random graph, then we assign propositions to the edges. To generate random graphs, we sampled uniformly from all directed graphs on ten vertices: each node is connected to each other node with probability 0.5 — in the Erdős-Rényi model, $G(10, 0.5)$ (Erdős and Rényi 1959).[6] Given

[6] Even though the graphs only contain ten nodes, computing reachability can still be computationally intensive, since we compute all simple paths and cycles.

Table 3. Speedups for each strategy for each bit vector logic in each solver on proposition graph reachability benchmarks

Benchmark	# SMT calls	Z3 PP	Z3 CSA	CVC4 PP	CVC4 CSA	Yices PP	Yices CSA	Boolector PP	Boolector CSA
QF_BV-1	7,984	0.85	**1.47**	1.01	**12.07**	0.98	0.87	1.03	**9.53**
QF_BV-2	1,994	0.95	**1.09**	1.06	**1.82**	0.98	0.96	0.97	**2.65**
QF_BV-3	9,088	0.86	**1.62**	1.11	**19.67**	0.98	1.00	1.10	**13.51**
QF_UFBV-1	19,729	0.90	**1.39**	1.06	**15.23**	0.96	**1.09**	1.10	**8.90**
QF_UFBV-2	16,145	0.88	**1.20**	1.10	**4.57**	0.97	**1.06**	1.14	**4.27**
QF_UFBV-3	18,137	0.87	**1.29**	1.21	**16.44**	0.98	**1.05**	1.15	**12.57**
QF_ABV-1	22,486	0.93	**1.09**	0.99	**2.56**	0.98	0.80	1.11	**41.97**
QF_ABV-2	59,713	0.91*	**1.28**	1.25	**3.16**	0.98	0.99	1.08	**26.08**
QF_ABV-3	12,284	0.92	**1.27**	1.09	**5.82**	0.98	**1.11**	1.00	**12.03**
QF_AUFBV-1	26,128	0.89	0.93	1.13	**4.81**	0.98	0.85	1.15	**7.55**
QF_AUFBV-2	19,136	0.92	**1.51**	1.06	**12.11**	0.97	**1.10**	1.10	**17.13**
QF_AUFBV-3	26,914	0.94	**1.52**	1.22	**13.41**	0.97	**1.05**	1.16	**27.08**
Average		0.90	**1.31**	1.11	**9.31**	0.98	0.99	1.09	**15.27**
Median		0.90	**1.29**	1.10	**8.94**	0.98	**1.02**	1.10	**12.30**

a graph, each edge is assigned a randomly generated formula. To generate a random formula of size m in a given theory, we randomly generate well-sorted applications of that theory's operators to random formulas of size $m/2$. The leaves of formulas are made up variables and constants; we generate variables twice as often as constants. We are careful to ensure that variables exist for every sort mentioned in the theory's operators. (For bit vectors and integers, we only generated constants in the range -50 to 50; we use both 4- and 8-bit widths for bit vectors.) We use Haskell's QuickCheck to generate random values (Claessen and Hughes 2000).

We used eight different SMT logics in our evaluations; for a given logic L, we generated three random graphs (named L-1, L-2, and L-3). Timeouts were five minutes.

We tested the logics of quantifier-free bit vectors (QF_BV), plus uninterpreted functions and/or arrays (QF_UFBV, QF_ABV, and QF_AUFBV), using the solvers Z3, CVC4, Yices, and Boolector (Table 3). For Z3, CVC4, and Boolector, the CSA strategy provided consistent speedups over the baseline. These speedups were relatively mild in the case of Z3 (an average of 1.31×), but substantial in the cases of CVC4 and Boolector (an average of 9.31× and 15.27×, respectively). For Yices, the CSA strategy was slightly better than the baseline about half the time and slightly worse the other half; on average, it performed on par with the baseline. The PP strategy was clearly not as effective on these benchmarks: it was consistently worse than the baseline for Z3 and Yices, and had relatively small speedups compared to the CSA strategy for CVC4 and Boolector (an average of 1.11× and 1.09×, respectively).

The story is more mixed for the logics of quantifier-free linear integer arithmetic (QF_LIA), with uninterpreted functions and/or arrays (QF_UFLIA, QF_ALIA, and QF_AUFLIA), which we tested using Z3, CVC4, and Yices (Table 4). For the array-free logics, the CSA strategy provided consistent, if mild, speedups to all the solvers (averages of 1.04×, 1.34×, and 1.03×, respectively). The strategy performed badly, though, for the logics with arrays, with a slight slowdown over the baseline for Z3, and substantial slowdowns for CVC4 and Yices (includ-

Table 4. Speedups for each strategy for each linear integer arithmetic logic in each solver on proposition graph reachability benchmarks

		Z3		CVC4		Yices	
Benchmark	# SMT calls	PP	CSA	PP	CSA	PP	CSA
QF_LIA-1	3,967	0.91	**1.05**	0.93	**1.91**	0.98	**1.01**
QF_LIA-2	2,307	0.99	1.00	0.99	**1.25**	0.96	1.00
QF_LIA-3	3,240	0.92	**1.01**	0.95	**1.12**	1.00	**1.01**
QF_UFLIA-1	16,856	0.88*	1.00	0.93	**1.16**	0.94	**1.03**
QF_UFLIA-2	13,200	0.86*	**1.09**	0.95	**1.26**	0.97	**1.02**
QF_UFLIA-3	44,154	N/A*	**1.11**	0.97	**1.34**	0.93	**1.13**
Average		0.91	**1.04**	0.95	**1.34**	0.96	**1.03**
Median		0.91	**1.03**	0.95	**1.26**	0.97	**1.02**
QF_ALIA-1	6,742	0.96	**1.08**	0.94	0.89	0.99	0.76
QF_ALIA-2	11,375	0.90	0.54	1.00	0.05	0.98	0.27
QF_ALIA-3	21,982	0.87	0.69	**1.02**	0.09	0.97	0.01†
QF_AUFLIA-1	18,416	0.90	0.92	0.97	0.18	0.99	0.36
QF_AUFLIA-2	7,190	0.97	**1.13**	0.97	0.49	0.98	0.03
QF_AUFLIA-3	19,037	0.90	**1.06**	0.97	0.50	0.96	0.15
Average		0.92	0.90	0.98	0.36	0.98	0.26
Median		0.90	0.99	0.97	0.33	0.98	0.21

ing consistent timeouts on the QF_ALIA-3 benchmark in the latter case). The PP strategy was consistently slightly worse than the baseline for both sets of benchmarks.

To better understand our results, we ran an additional set of experiments using the same proposition graphs (Table A 4). In these experiments, a program computes reachability from a single distinguished node using a search similar to DFS. Incremental solving gave speedups on 76 of 84 benchmarks (90%). On 54 benchmarks (64%), the PP strategy outperformed the CSA strategy, suggesting that for workloads with the right solver locality, it can often provide improved performance. This indicates that our results for parallel all-pairs reachability, where CSA is consistently the better strategy, really reflect the fact that the programs do not have good solver locality for PP (matching our intuition that CSA would be a better fit for non-DFS settings). In these additional experiments, we also found that the incremental strategies often provided speedups in the benchmarks combining linear integer arithmetic and arrays, where we previously had no speedups. This indicates that there is the potential for gains from incremental solving for these logics, but our strategies are not delivering them for the all-pairs reachability workload; this could perhaps be more of a reflection of current solver technology than anything fundamental.

4.4 Summary

Our results confirm our two hypotheses:

There is a gain to be had from incremental solving, despite the lack of obvious solver locality under evaluation strategies like Formulog's. Across all case studies, we ran 105 distinct experiments; on 81 of these experiments (77%), either the PP strategy or CSA strategy led to some speedup over the non-incremental baseline. Furthermore, of the 24 counterexamples, most of them (14) are among the proposition graph reachability experiments with logics combining linear integer arithmetic and arrays; these logics seem to be the exception rather than the norm.

Our numbers demonstrate that CSA strategy is more effective than the PP strategy. The CSA strategy provided a speedup on 79 experiments, compared to 39 for the PP strategy. There were only two instances where the PP strategy provided a speedup and the CSA strategy did not, and only five instances where both strategies led to speedups but the PP strategy was faster.

5 Limitations and related work

In an empirical evaluation involving SMT solving, the generality of any result is necessarily limited, as it reflects only current solver technology. SMT solvers are constantly evolving, and new heuristics, for example, could dramatically change the performance of a strategy. Nevertheless, we found relatively consistent and robust results across a variety of benchmarks, SMT solvers, and SMT logics. We have a reasonable basis to believe that our results should generalize beyond the context of our evaluation.

This paper has assumed a black-box interaction between the logic programming system and SMT solver. It might be possible to more tightly integrate the two, passing more information between them that could improve incremental solving performance. Previous work has explored this type of integration within SMT solvers (Nieuwenhuis et al. 2006) and constraint answer set programming solvers (Balduccini and Lierler 2013). However, there are good arguments for black-box usage of an SMT solver, not least of all is the ability to pull one off the shelf and use it without substantial changes to the logic programming runtime or SMT solver itself.

Furthermore, we have only evaluated lightweight mechanisms for achieving incremental solving. Many alternative mechanisms are possible, and could be worth exploring in future work. For example, a heavier-weight mechanism could maintain a pool of SMT solvers, and given an SMT query, assign it to a solver in a way to maximize incremental solving, or try different strategies simultaneously on different solvers, accepting the answer of the first one that finishes, like portfolio solving (Wintersteiger et al. 2009). Along a different dimension, it might be possible for a logic programming language to more directly expose an incremental SMT solving interface to the programmer (albeit this might be less in the spirit of Kowalski's principle).

Other work has explored how to make effective use of incremental SMT solving for particular applications, such as bounded model checking (Günther and Weissenbacher 2014) and symbolic execution (Liu et al. 2014). Our approach is agnostic to the application, but does not address the question of how an application can generate constraints with shared conjuncts (a prerequisite for our approach that is not always easy to achieve).

6 Conclusion

This paper has explored whether a logic programming system can take advantage of incremental solving in an external SMT solver, even when the former does not generate SMT queries in an order that has obviously good "solver locality" and the latter is used only as a black-box. Across a range of benchmarks, SMT solvers, and SMT logics, we evaluated two different strategies that the logic programming system Formulog uses to pose SMT queries to the solver. A strategy based on the `check-sat-assuming` command consistently led to speedups over a non-incremental baseline, demonstrating that it is possible to take advantage of incremental SMT solving in a logic programming setting: one needs to merely maintain a lightweight map from formulas to assumption variables to mediate between the logic programming runtime and SMT solver.

References

BALDUCCINI, M. AND LIERLER, Y. 2013. Integration schemas for constraint answer set programming: a case study. *Theory and Practice of Logic Programming 13,* 4-5, 1–12.

BANCILHON, F. 1986. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems*. 165–178.

BARRETT, C., CONWAY, C. L., DETERS, M., HADAREAN, L., JOVANOVIĆ, D., KING, T., REYNOLDS, A., AND TINELLI, C. 2011. CVC4. In *Proc. 23rd Int. Conf. on Computer Aided Verification*. 171–177.

BARRETT, C., FONTAINE, P., AND TINELLI, C. 2017. The SMT-LIB standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa.

BEMBENEK, A., GREENBERG, M., AND CHONG, S. 2020. Formulog: Datalog for SMT-based static analysis. In submission.

BIERMAN, G. M., GORDON, A. D., HRIŢCU, C., AND LANGWORTHY, D. 2012. Semantic subtyping with an SMT solver. *Journal of Functional Programming 22,* 1, 31–105.

CADAR, C. AND SEN, K. 2013. Symbolic execution for software testing: Three decades later. *Commun. ACM 56,* 2 (Feb.), 82–90.

CIMATTI, A. AND GRIGGIO, A. 2012. Software model checking via IC3. In *Proc. 24th Int. Conf. on Computer Aided Verification*. 277–293.

CLAESSEN, K. AND HUGHES, J. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. 5th ACM SIGPLAN Int. Conf. on Functional Programming*. 268–279.

DE MOURA, L. AND BJØRNER, N. 2008. Z3: An efficient SMT solver. In *Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

DUTERTRE, B. 2014. Yices 2.2. In *In Proc. 26th Int. Conf. on Computer-Aided Verification*. 737–744.

EÉN, N. AND SÖRENSSON, N. 2003. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science 89,* 4, 543–560.

ERDŐS, P. AND RÉNYI, A. 1959. On random graphs I. *Publ. Math. 6*, 290–297.

GÜNTHER, H. AND WEISSENBACHER, G. 2014. Incremental bounded software model checking. In *Proc. 2014 Int. Symp. on Model Checking of Software*. 40–47.

JHA, S., GULWANI, S., SESHIA, S. A., AND TIWARI, A. 2010. Oracle-guided component-based program synthesis. In *Proc. 32nd ACM/IEEE Int. Conf. on Software Engineering*. 215–224.

KING, J. C. 1976. Symbolic execution and program testing. *Commun. ACM 19,* 7, 385–394.

KOWALSKI, R. 1979. Algorithm = logic + control. *Commun. ACM 22,* 7, 424–436.

LATTNER, C. AND ADVE, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. 2nd IEEE/ACM Int. Symp. on Code Generation and Optimization*. 75–88.

LEINO, K. R. M. 2010. Dafny: An automatic program verifier for functional correctness. In *Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

LIU, T., ARAÚJO, M., D'AMORIM, M., AND TAGHDIRI, M. 2014. A comparative study of incremental constraint solving approaches in symbolic execution. In *Haifa Verification Conference*. 284–299.

NIEMETZ, A., PREINER, M., AND BIERE, A. 2014 (published 2015). Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation 9*, 53–58.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM (JACM) 53,* 6, 937–977.

WINTERSTEIGER, C. M., HAMADI, Y., AND DE MOURA, L. 2009. A concurrent portfolio approach to SMT solving. In *Proc. 21st Int. Conf. on Computer Aided Verification*. 715–720.

Appendix A Additional tables

Table A 1. Time (s) for each strategy in each solver on symbolic execution benchmarks

	Z3			CVC4		
Benchmark	Baseline	PP	CSA	Baseline	PP	CSA
shuffle-4	5	**3**	**3**	33	**6**	7
shuffle-5	54	19	**17**	$1800^{\dagger}$	**135**	167
sort-6	56	62	**29**	$1800^{\dagger}$	$1800^{\dagger}$	**30**
sort-7	391	434	**228**	$1800^{\dagger}$	$1800^{\dagger}$	**234**
numbrix-sat	55	**36**	49	83	**78**	92
numbrix-unsat	46	**31**	41	64	**49**	59

Table A 1 (cont.). Time (s) for each strategy in each solver on symbolic execution benchmarks

	Boolector		
Benchmark	Baseline	PP	CSA
shuffle-4	733	46	**44**
shuffle-5	$1800^{\dagger}$	1434	**614**
sort-6	404	345	**34**
sort-7	$1800^{\dagger}$	$1800^{\dagger}$	**233**
numbrix-sat	202	**119**	**119**
numbrix-unsat	186	124	**120**

Table A 2. Times (s) for each strategy on refinement type checking benchmarks

Benchmark	Baseline	PP	CSA
all-1	10	**5**	**5**
all-10	319	29	**23**
all-100	$3600^{\dagger}$	299	**202**

Table A 3. Time (ms) for each strategy for each logic in each solver on proposition graph reachability benchmarks

		Z3			CVC4	
Benchmark	Baseline	PP	CSA	Baseline	PP	CSA
QF_BV-1	3131	3684	**2128**	27288	26952	**2252**
QF_BV-2	1327	1399	**1216**	2302	2162	**1267**
QF_BV-3	4080	4750	**2514**	47395	42525	**2409**
QF_UFBV-1	2688	2988	**2514**	36054	34115	**2367**
QF_UFBV-2	1807	2051	**1506**	7457	6773	**1630**
QF_UFBV-3	2109	2412	**1631**	32923	27315	**2003**
QF_ABV-1	3677	3971	**3371**	87534	88756	**34229**
QF_ABV-2	6664	7332*	**5198**	175831	140875	**55719**
QF_ABV-3	2205	2389	**1739**	23648	21657	**4065**
QF_AUFBV-1	3596	4020	3860	30930	27264	**6436**
QF_AUFBV-2	2856	3114	**1893**	55150	52046	**4554**
QF_AUFBV-3	4324	4609	**2845**	95692	78728	**7134**
QF_LIA-1	1494	1649	**1429**	2966	3198	**1555**
QF_LIA-2	1185	1201	1188	1454	1465	**1166**
QF_LIA-3	1199	1306	**1188**	1282	1347	**1149**
QF_UFLIA-1	1666	1903*	1667	2215	2375	**1908**
QF_UFLIA-2	1628	1898*	**1492**	2359	2492	**1866**
QF_UFLIA-3	2764	N/A*	**2494**	5492	5681	**4085**
QF_ALIA-1	1499	1562	**1382**	1743	1860	1965
QF_ALIA-2	2016	2232	3765	4315	4337	88495
QF_ALIA-3	2343	2688	3409	7287	**7134**	76707
QF_AUFLIA-1	2017	2234	2192	4213	4321	24004
QF_AUFLIA-2	1785	1838	**1577**	4032	4176	8290
QF_AUFLIA-3	2133	2373	**2010**	4065	4202	8193

Table A 3 (cont.). Time (ms) for each strategy for each logic in each solver on proposition graph reachability benchmarks

	Yices			Boolector		
Benchmark	Baseline	PP	CSA	Baseline	PP	CSA
QF_BV-1	1601	1629	1842	41681	40570	**4375**
QF_BV-2	1079	1101	1126	6103	6297	**2301**
QF_BV-3	1820	1851	1827	42880	38867	**3173**
QF_UFBV-1	1903	1976	**1740**	39330	35823	**4418**
QF_UFBV-2	1447	1488	**1362**	7028	6187	**1645**
QF_UFBV-3	1652	1693	**1569**	36019	31422	**2866**
QF_ABV-1	2176	2211	2703	221540	199575	**5279**
QF_ABV-2	3826	3929	3876	220352	203139	**8448**
QF_ABV-3	1792	1821	**1616**	44170	44061	**3672**
QF_AUFBV-1	2163	2196	2547	85078	74113	**11276**
QF_AUFBV-2	1996	2064	**1809**	60588	55272	**3538**
QF_AUFBV-3	2766	2842	**2644**	156223	135087	**5770**
QF_LIA-1	1207	1237	**1190**			
QF_LIA-2	1081	1126	**1077**			
QF_LIA-3	1125	1126	**1111**			
QF_UFLIA-1	1380	1467	**1337**			
QF_UFLIA-2	1318	1355	**1298**			
QF_UFLIA-3	1893	2032	**1679**			
QF_ALIA-1	1274	1285	1674			
QF_ALIA-2	1524	1556	5544			
QF_ALIA-3	1863	1918	300000[†]			
QF_AUFLIA-1	1691	1714	4647			
QF_AUFLIA-2	1523	1552	50876			
QF_AUFLIA-3	1843	1911	12470			

Table A 4. Speedup for each strategy for each logic in each solver on single-origin, DFS-like proposition graph reachability benchmarks

Benchmark	# SMT calls	Z3 PP	Z3 CSA	CVC4 PP	CVC4 CSA	Yices PP	Yices CSA	Boolector PP	Boolector CSA
QF_BV-1	1798	1.05	**1.06**	1.32	**1.41**	**1.02**	0.97	1.46	**2.25**
QF_BV-2	293	0.98	0.98	**1.01**	0.99	1.00	0.97	**1.28**	1.15
QF_BV-3	966	1.05	**1.07**	1.28	**1.44**	**1.03**	1.01	**1.66**	1.65
QF_UFBV-1	4591	1.16	**1.18**	2.82	**3.70**	1.06	**1.08**	2.56	**5.33**
QF_UFBV-2	1388	0.99	**1.01**	**1.12**	1.11	0.99	0.98	**1.06**	1.02
QF_UFBV-3	2088	**1.04**	1.02	1.34	**1.49**	**1.05**	1.02	1.64	**2.47**
QF_ABV-1	2584	**1.15**	1.08	**1.58**	0.96	**1.12**	1.05	2.46	**3.30**
QF_ABV-2	8360	1.24	**1.25**	**2.64**	2.46	**1.14**	1.12	3.10	**6.64**
QF_ABV-3	1086	**1.04**	1.01	**1.17**	1.16	**1.04**	1.00	1.07	**1.67**
QF_AUFBV-1	4530	**1.13**	1.09	2.03	**2.19**	**1.15**	1.12	2.80	**4.22**
QF_AUFBV-2	3403	**1.12**	**1.12**	2.37	**2.82**	1.08	**1.09**	2.47	**4.23**
QF_AUFBV-3	7002	**1.30**	1.29	3.30	**4.70**	**1.21**	1.13	3.75	**9.11**
QF_LIA-1	440	**1.04**	0.99	**1.04**	0.99	**1.03**	0.98		
QF_LIA-2	395	1.00	1.00	0.97	0.97	0.99	0.97		
QF_LIA-3	480	0.93	0.94	0.97	0.95	**1.02**	0.97		
QF_UFLIA-1	1874	**1.04**	1.00	**1.07**	1.02	**1.03**	1.00		
QF_UFLIA-2	2689	1.02	**1.04**	1.04	**1.05**	**1.03**	**1.03**		
QF_UFLIA-3	3990	**1.06**	1.03	**1.16**	1.12	**1.07**	**1.07**		
QF_ALIA-1	1444	0.99	**1.07**	**1.03**	1.00	**1.04**	1.01		
QF_ALIA-2	2197	**1.09**	0.97	**1.19**	0.45	**1.10**	1.05		
QF_ALIA-3	2831	**1.10**	1.03	**1.24**	0.69	**1.07**	0.91		
QF_AUFLIA-1	2435	**1.05**	1.03	**1.12**	0.88	**1.06**	**1.01**		
QF_AUFLIA-2	710	0.96	**1.03**	**1.05**	0.96	0.98	0.95		
QF_AUFLIA-3	6320	**1.16**	1.14	**1.30**	0.84	**1.17**	0.96		
Average		**1.07**	1.06	**1.47**	**1.47**	**1.06**	1.02	2.11	**3.59**
Median		**1.05**	1.03	**1.18**	1.03	**1.05**	1.01	2.06	**2.89**

Table A 5. Time (ms) for each strategy for each logic in each solver on single-origin, DFS-like proposition graph reachability benchmarks

		Z3			CVC4	
Benchmark	Baseline	PP	CSA	Baseline	PP	CSA
QF_BV-1	1232	1174	**1166**	1744	1325	**1237**
QF_BV-2	968	983	986	971	**959**	982
QF_BV-3	1198	1143	**986**	1609	1255	**1119**
QF_UFBV-1	1368	1180	**1164**	4690	1666	**1267**
QF_UFBV-2	1067	1077	**1056**	1160	**1038**	1046
QF_UFBV-3	1095	**1048**	1074	1663	1242	**1114**
QF_ABV-1	1340	**1169**	1246	4408	**2781**	4598
QF_ABV-2	1897	1526	**1512**	8144	**3085**	3304
QF_ABV-3	1081	**1044**	1066	1339	**1147**	1155
QF_AUFBV-1	1461	**1289**	1336	3734	1841	**1708**
QF_AUFBV-2	1305	**1164**	1169	3832	1614	**1359**
QF_AUFBV-3	1797	**1381**	1187	10160	3078	**2160**
QF_LIA-1	994	**954**	1003	990	**951**	1002
QF_LIA-2	935	937	**934**	944	969	978
QF_LIA-3	917	990	980	915	939	961
QF_UFLIA-1	1071	**1034**	1068	1073	**1002**	1051
QF_UFLIA-2	1127	1109	**1087**	1147	1102	**1093**
QF_UFLIA-3	1159	**1092**	1127	1283	**1104**	1151
QF_ALIA-1	1106	1112	**1037**	1101	**1074**	1100
QF_ALIA-2	1245	**1147**	1278	1389	**1165**	3106
QF_ALIA-3	1223	**1108**	1192	1505	**1216**	2191
QF_AUFLIA-1	1163	**1105**	1124	1298	**1162**	1473
QF_AUFLIA-2	1073	1122	**1044**	1133	**1080**	1180
QF_AUFLIA-3	1478	**1270**	1295	1864	**1432**	2209

Table A 5 (cont.). Time (ms) for each strategy for each logic in each solver on single-origin, DFS-like proposition graph reachability benchmarks

	Yices			Boolector		
Benchmark	Baseline	PP	CSA	Baseline	PP	CSA
QF_BV-1	1069	**1053**	1102	4010	2752	**1785**
QF_BV-2	929	932	955	1378	**1074**	1199
QF_BV-3	1042	**1008**	1029	3102	**1864**	1878
QF_UFBV-1	1201	1128	**1112**	7577	2961	**1421**
QF_UFBV-2	1006	1013	1031	1384	**1310**	1359
QF_UFBV-3	1066	**1014**	1044	3532	2157	**1429**
QF_ABV-1	1233	**1106**	1174	6422	2615	**1944**
QF_ABV-2	1520	**1337**	1357	16080	5192	**2421**
QF_ABV-3	1050	**1010**	1046	2140	1997	**1283**
QF_AUFBV-1	1337	**1159**	1192	9513	3400	**2254**
QF_AUFBV-2	1219	1132	**1118**	7403	2994	**1748**
QF_AUFBV-3	1518	**1253**	1341	19133	5108	**2100**
QF_LIA-1	959	**933**	982			
QF_LIA-2	937	950	962			
QF_LIA-3	947	**927**	973			
QF_UFLIA-1	1039	**1008**	1038			
QF_UFLIA-2	1083	1056	**1052**			
QF_UFLIA-3	1139	**1062**	1065			
QF_ALIA-1	1068	**1023**	1058			
QF_ALIA-2	1190	**1082**	1139			
QF_ALIA-3	1175	**1096**	1296			
QF_AUFLIA-1	1141	**1081**	1125			
QF_AUFLIA-2	1057	1081	1114			
QF_AUFLIA-3	1481	**1271**	1547			