# Incremental Rewriting Modulo SMT

Gerald Whitters[1], Boon Thau Loo[1], Vivek Nigam[3,4], and Carolyn Talcott[2]

[1] UPENN, Philadelphia, USA, {whitters,boonloo}@seas.upenn.edu
[2] SRI International, Menlo Park, USA, carolyn.talcott@gmail.com
[3] Federal University of Paraíba, João Pessoa, Brazil, vivek.nigam@gmail.com
[4] Huawei Munich Research Center, Germany

**Abstract.** Rewriting Modulo SMT combines two powerful automated deduction techniques (1) rewriting and (2) SMT-solving. Rewriting enables the specification of behavior of systems using rewriting rules, while SMT theories specify system properties. Rewriting Modulo SMT is enabled by combining existing tools, such as Maude and SMT-solvers. Search algorithms used for carrying out Rewriting Modulo SMT, however, cannot exploit the incremental solving features available in SMT-solvers as they are based on breadth-first search. This paper addresses this limitation by proposing Incremental Rewriting Modulo SMT Theories, which is a syntactical restriction to rewriting rules. This restriction turns out to naturally be used in several applications of Rewriting Modulo SMT, including the verification of algorithms, cyber-physical systems, and security protocols. Moreover, we propose a Hybrid-Search algorithm for Incremental Rewriting Modulo SMT Theories that combines breadth-first search and depth-first search thus enabling incremental SMT-solving. We demonstrate through a collection of existing benchmarks that the Hybrid-Search algorithm can achieve a 10X performance improvement in verification times.

## 1 Introduction

Rewriting modulo SMT [14] is the result of the combination of two powerful automated deduction methods: rewriting logic and SMT-solving. It is supported by the integration [11] of powerful tools, such as Maude [6] and Z3 [8]. During rewriting, a set of constraints on the symbols appearing in a term are generated. These constraints can be, for example, non-linear arithmetic constraints that specify possible values that can be assumed by the configuration parameters. Demonstrating properties of such specifications amounts to search using these rewrite rules and satisfiability checking of the accumulated constraints using SMT-solvers. Rewriting modulo SMT has been successfully applied in several case-studies from several domains, including safety of cyber-physical systems (CPSes) [13]; verification of algorithms [2]; and for network security analysis [16].

One important aspect that has not been addressed until now is how to exploit an SMT-solver's capability of incrementally solving problems. In this solving method, instead of checking for the satisfiability of a formula from scratch, it reuses data previously computed by prior checks. For example, if the satisfiability

of a formula $\mathsf{b}$ has been checked, the check on $\mathsf{b} \wedge \mathsf{b}_I$ may re-use the intermediate results obtained while checking for the satisfiability of $\mathsf{b}$. It has been shown that incremental solving can greatly improve performance by a factor of 2-5 times [10].[5]

The search algorithms used to implement rewriting modulo SMT are similar to those implemented in the Maude search engine [6]. They use a breadth-first search (BFS) algorithm with memoization techniques in order to improve performance. This type of search seems incompatible with incremental solving as constraints appearing in different branches of the search tree are generated under different conditions. Thus, it is hard to define what the increment ($\mathsf{b}_I$ mentioned above) would be.

This paper's goal is to enable rewriting modulo SMT that can exploit incremental solving. To achieve this, we make the following contributions:

- **Incremental Rewriting Modulo SMT** by identifying a class of rewrite rules that are amenable to incremental solving. More specifically, rewrite rules are applied to terms containing symbols paired with a set of boolean terms constraining the values of these symbols. Moreover, any rewrite rule can only add new constraints, i.e.not change the existing set of constraints on the term that is being rewritten. We show that a variety of theories used in published case studies can be seen amenable to incremental solving.
- **A Hybrid Search Algorithm for Incremental Theories** which combines breadth and depth-first search (DFS) strategies. The combination is parameterized by a level of depth parameter which specifies how many depth-first search steps shall be performed before switching to a breath-first search. The proposed hybrid search algorithm enjoys the benefits of BFS, namely better coverage as it alternates through different branches of the search tree, and the benefits of DFS, namely incremental solving.

We carried out a collection of experiments (the case studies mentioned above) on algorithm verification, cyber-physical systems verification, and network security analysis. The experiments show that in all these benchmarks, the hybrid search algorithm out-performs current BFS techniques, in some experiments achieving a 10 factor performance improvement.

Section 2 illustrates the problems of existing BFS methods for Rewriting Modulo SMT and proposes Incremental Rewriting Theories which formalizes the notion of increments. Section 3 describes the Hybrid algorithm proposed illustrating how it enables incremental SMT solving. Section 4 describes experiments that compare different search mechanisms (BFS, DFS, and Hybrid) on existing benchmarks from the literature. Finally, we conclude by discussing Related Work in Section 5 and Future Work in Section 6.

[5] Albeit, incremental solving can also reduce performance depending on the theories that are used.

## 2 Incremental Rewriting Modulo SMT

Rewriting logic [12] is a logical formalism that is based on two ideas: states of a system are represented as elements of an algebraic data type, specified in an equational theory, and the behavior of a system is given by local transitions between states described by rewrite rules. A rewrite rule has the form $\mathsf{t} \rightarrow \mathsf{t}'$ if $\mathsf{b}$, where $\mathsf{t}$ and $\mathsf{t}'$ are terms possibly containing variables and $\mathsf{b}$ is a condition (a boolean term). Such a rule applies to a system in state $\mathsf{s}$ (a ground term) if $\mathsf{t}$ can be matched to a part of $\mathsf{s}$ by supplying the right values for the variables, and if the condition $\mathsf{b}$ holds when supplied with those values. In this case, the rule can be applied by replacing the part of $\mathsf{s}$ matching $\mathsf{t}$ by $\mathsf{t}'$ using the matching values for the variables in $\mathsf{t}'$.

Maude is a language and tool based on rewriting logic [6]. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. Thus, given a specification $S$ of a concurrent system, one can execute $S$ to find one possible behavior; use search to see if a state meeting a given condition can be reached; or model-check $S$ to see if a temporal property is satisfied, and if not, to see a computation that is a counter example.

Symbolic rewriting modulo SMT [14,13] allows rewriting symbolic states $(\mathsf{t}, \mathsf{b})$, where $\mathsf{t}$ is a term possibly containing variables and $\mathsf{b}$ a boolean term constraining the allowed values of variables of $\mathsf{t}$. To implement this in Maude, variables are replaced by symbols, treated as constants by Maude, and translated as variables when using an SMT solver to check satisfiability of the constraint. Symbolic rewriting allows us to reason about open systems, and to reason about all (possibly infinitely many) instances of a configuration.

Verification problems are expressed as reachability problems expressed as statements of the form

$$\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow (\mathsf{t}', \mathsf{b}') \text{ such that } \mathsf{goalCond}(\mathsf{t}', \mathsf{b}')$$

where $(\mathsf{t}', \mathsf{b}')$ is a pattern and $\mathsf{goalCond}$ is a boolean function that checks whether a state satisfies some condition. Typically, $\mathsf{goalCond}(\mathsf{t}', \mathsf{b}')$ also makes calls to the SMT-solver to check whether some constraints derived from $\mathsf{b}'$ are satisfiable.

As illustrated by Figure 1, Rewriting Modulo SMT implementations [11] traverse the search tree derived from the rewrite rules using BFS-based algorithms. At each step, e.g., $(\mathsf{t}_0, \mathsf{b}_0) \rightarrow (\mathsf{t}_1, \mathsf{b}_1)$, the engine checks for the satisfiability of the condition $\mathsf{b}_1$. If the check fails, then search backtracks following BFS strategy. Otherwise, if the check succeeds, then the engine checks (1) whether $(\mathsf{t}_1, \mathsf{b}_1)$ matches the pattern $(\mathsf{t}', \mathsf{b}')$ and (2) if this is the case, it checks the condition $\mathsf{goalCond}(\mathsf{t}_1, \mathsf{b}_1)$, which may make further calls to the SMT-solver, written as $\mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_1, \mathsf{b}_1))$. If $\mathsf{goalCond}$ returns true, then a solution for the reachability problem is found. Otherwise, the algorithm continues search following BFS.

From the sequence of calls to the SMT-solver, one can observe the following difficulties of exploiting incremental SMT solving when using BFS based search strategy:

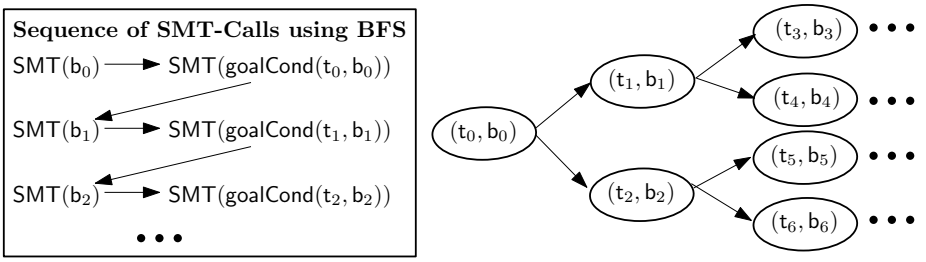


**Fig. 1.** Illustration of the search tree and SMT-calls when using Rewriting Modulo SMT following a BFS algorithm. The sequence of SMT-calls of a BFS algorithm is depicted to the left, where $\mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_i, \mathsf{b}_i))$ denotes possible SMT-calls required by the goal condition goalCond.

- **Definitions of Increments:** Given the generality of the accepted theory, it is not possible for the search engine to determine whether constraints, $\mathsf{b}_1$ and $\mathsf{b}_2$, used in subsequent calls to the SMT, $\mathsf{SMT}(b_1)$ and $\mathsf{SMT}(b_2)$, are constructed using some increment, i.e., whether $\mathsf{b}_2 = \mathsf{b}_1 \wedge \mathsf{b}_{1,2}$. This is because $\mathsf{b}_1$ and $\mathsf{b}_2$ are derived by applying different instances of rules which normally add/modify constraints in different ways.
- **Not possible to chain incremental calls:** As it is not possible to define increments when using rewrites rules in general, it is not possible to effectively use incremental solving by chaining calls, such as in $\mathsf{SMT}(\mathsf{b}_1); \mathsf{SMT}(\mathsf{b}_1 \wedge \mathsf{b}_{1,2}); \mathsf{SMT}(\mathsf{b}_1 \wedge \mathsf{b}_{1,2} \wedge \mathsf{b}_{2,3}) \ldots$.

To address this problem, we introduce a special class of rewrite theories, called *Incremental Rewrite Theories*.

**Definition 1.** *An incremental rewrite theory is a rewrite theory specification $\langle \Sigma, \mathcal{E}, \mathcal{R} \rangle$ where $\Sigma$ is a typed alphabet; $\mathcal{E}$ is an equational theory; and $\mathcal{R}$ is a set of rewrite rules of the forms:*

$$(\mathsf{t}, \mathsf{b}) \rightarrow (\mathsf{t}_1, \mathsf{b} \wedge \mathsf{b}_I) \qquad \textit{and} \qquad (\mathsf{t}, \mathsf{b}) \rightarrow (\mathsf{t}_1, \mathsf{b} \wedge \mathsf{b}_I) \textit{ if } \mathsf{cond}(\mathsf{t})$$

*where $\mathsf{t}$ is a well-formed term; $\mathsf{b}, \mathsf{b}_I$ are boolean formulas (in a given theory); and cond is a function that takes a term $\mathsf{t}$ and returns a boolean value.*

The verification problem for incremental problems is a specialized reachability problem as defined below.

**Definition 2.** *Let $\mathcal{T}$ be an incremental rewrite theory. An incremental reachability problem over $\mathcal{T}$ is of the form:*

$$\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow (\mathsf{t}', \mathsf{b}') \textit{ such that } \mathsf{goalTerm}(\mathsf{t}') \textit{ and } \mathsf{SMT}(\mathsf{b}' \wedge \mathsf{b}_I)$$

*where goalTerm is a function that takes a term and returns a boolean value and $\mathsf{b}_I = \mathsf{goal}(\mathsf{t}')$ is a formula constructed from $\mathsf{t}'$.*

The following three examples illustrate how incremental theories can model different types of systems. These examples are based on specifications from the literature [2,16,13]. For ease of exposition, we simplify the rules in the description below. In Section 4, the full specifications from the literature is used in our experiments.

*Example 1.* This example is based on the work [2] for verification of the CASH scheduling algorithm [4]. In this algorithm, each task has a worst-case execution time. Whenever a task is completed before its deadline, the unused processing time is added to a global queue of unused budget, which can then be used by other tasks. Rewriting modulo SMT has been used to verify whether it is possible for a task to miss its deadline [2]. In particular, constraints keep track of the processing times and the available time budgets.

It turns out that the specification of this algorithm as rewrite rules and the verification problem is an incremental rewrite theory. For example, the following rule specifies when a deadline is missed:

$$
\begin{array}{l}
(\langle \mathsf{id}_1 : \mathsf{global} \mid \mathsf{deadlineMiss} : \mathsf{b}', \mathsf{Ats} \rangle, \\
\langle \mathsf{id}_0 : \mathsf{server} \mid \mathsf{state} : st, \mathsf{usedBudget} : t, \mathsf{timeDeadline} : t_1, \mathsf{maxBudget} : n \rangle \ \mathsf{rest}, \mathsf{b}) \\
\rightarrow (\langle \mathsf{id}_1 : \mathsf{global} \mid \mathsf{deadlineMiss} : \mathsf{true}, \mathsf{Ats} \rangle) \\
(\langle \mathsf{id}_0 : \mathsf{server} \mid \mathsf{state} : st, \mathsf{usedBudget} : t, \mathsf{timeDeadline} : t_1, \mathsf{maxBudget} : n \rangle \ \mathsf{rest}, \\
\mathsf{b} \wedge \mathsf{b}_I) \ \mathsf{if} \ (st = \mathsf{waiting} \vee st = \mathsf{executing})
\end{array}
$$

where $\mathsf{rest}$ is the specification of the remaining tasks, $\mathsf{Ats}$ are other attributes of the server, $\mathsf{b}_I$ is the set of constraints $t \geq 0 \wedge t_1 \geq 0 \wedge n > 0 \wedge (n - t) > t_1$. This rule specifies that the deadline is missed if there is a task $\mathsf{id}_0$ that is not finished, i.e., either waiting or executing, such that the time to finish ($t_1$) cannot be met by the available time budget $n - t$ required by the task.

The verification problem of checking whether for some given configuration $(\mathsf{t}_0, \mathsf{b}_0)$ of server and tasks, a task can miss its deadline is specified by the search command

$$
\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow (\langle \mathsf{id}_1 : \mathsf{global} \mid \mathsf{deadlineMiss} : \mathsf{true}, \mathsf{Ats} \rangle \ \mathsf{rest}, \mathsf{b}') \ \mathsf{such\ that}\ \mathsf{SMT}(\mathsf{b}')
$$

*Example 2.* Rewriting Modulo SMT has been used for verifying whether resource bounded intruders can slowly deny access to web-servers [16]. This type of attack was inspired by application layer DDoS attacks such as Slowloris [7] where the attacker attempts to exhaust all the resources of a web-server by periodically sending bursts of multiple requests. When receiving such bursts of requests, the web-server has to allocate resources for at least some period of time, called timeout. As the web-server has limited resources, the attacker is capable of denying service to legitimate users by sending enough bursts.

Constraints were used in reference [16] to keep track of (1) the number of resources available by the web-servers, and (2) the timeout period of bursts. While we refer to reference [16] for the complete formalization, we illustrate the incrementality of such specifications with a simplified version of the protocol

initialization rule from reference [16].

$$([\mathsf{iid} \mid \mathsf{pxs} \mid \mathsf{ri} \mid \mathsf{Trec}]\ [\mathsf{sid} \mid \mathsf{pxs}' \mid \mathsf{rs}], \mathsf{b}) \rightarrow$$
$$([\mathsf{iid} \mid \mathsf{px}(\mathsf{num}, \mathsf{rp})\ \mathsf{pxs} \mid \mathsf{ri}^{\nu} \mid \mathsf{Trec}]\ [\mathsf{sid} \mid \mathsf{px}(\mathsf{num}, \mathsf{rp})\ \mathsf{pxs}' \mid \mathsf{rs}^{\nu}], \mathsf{b} \wedge \mathsf{b}_I)$$

This rule specifies that the intruder iid with ri resources creates a new burst of protocol session instances $\mathsf{px}(\mathsf{num}, \mathsf{rp})$ with num instances each using rp resources, where num is a symbol. These instance requests are received by the server sid which has rs resources. The resources of the intruder, ri, and the resources of the server rs are updated to the fresh symbols $\mathsf{ri}^{\nu}$ and $\mathsf{rs}^{\nu}$. These symbols are constrained by the boolean increment $\mathsf{b}_I$ defined as $\mathsf{ri}^{\nu} = (\mathsf{ri} - \mathsf{num} \times \mathsf{rp}) \wedge \mathsf{rs}^{\nu} = (\mathsf{rs} - \mathsf{num} \times \mathsf{rp}) \wedge \mathsf{num} > 0 \wedge \mathsf{ri}^{\nu} \geq 0$. Similar rules specify when the protocol sessions timeout and are cleaned up by the server thus releasing resources.

The verification property is to check whether a bounded intruder with some limited number of resources ri can deny service by consuming the server sid's resources. This can be expressed by an incremental reachability property as follows where $(\mathsf{t}_0, \mathsf{b}_0)$ specifies the initial condition when all intruder and server resources are free:

$$\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow ([\mathsf{iid} \mid \mathsf{pxs} \mid \mathsf{ri} \mid \mathsf{Trec}]\ [\mathsf{sid} \mid \mathsf{pxs}' \mid \mathsf{rs}], \mathsf{b}')\ \mathsf{such\ that}\ \mathsf{SMT}(\mathsf{b}' \wedge \mathsf{b}_I)$$

where $\mathsf{b}_I$ is the constraint $\mathsf{rs} \leq 0$ specifying that the resources of the server sid are depleted.

*Example 3.* This example of verification of cyber-physical systems (CPSes) is based on reference [13]. A CPS is represented by a set of agents $(\mathsf{ag}_1, \ldots, \mathsf{ag}_n)$ that interact with the environment (env) to achieve some goal while not violating properties, such as the minimum distance to other objects.

Constraints are used to specify agent's physical attributes, such as its position, $\mathsf{at}(\mathsf{ag}, (x, y))$, speed, $\mathsf{spd}(\mathsf{ag}, v)$, acceleration, $\mathsf{acc}(\mathsf{ag}, acc)$, and direction $\mathsf{dir}(\mathsf{ag}, dir)$ of an agent ag. The evolution of a system with one agent can be specified by the following incremental rule when assuming, for simplicity, that the agent's direction is on the x-axis.

$$([\mathsf{env} \mid \mathsf{at}(\mathsf{ag}, (x, y)), \mathsf{spd}(\mathsf{ag}, v), \mathsf{acc}(\mathsf{ag}, acc), \mathsf{dir}(\mathsf{ag}, dir), \mathsf{kb}]\ \mathsf{conf}, \mathsf{b}\} \rightarrow$$
$$\{[\mathsf{env} \mid \mathsf{at}(\mathsf{ag}, (x_1, y_1)), \mathsf{spd}(\mathsf{ag}, v_1), \mathsf{acc}(\mathsf{ag}, acc), \mathsf{dir}(\mathsf{ag}, dir), \mathsf{kb}]\ \mathsf{conf}, \mathsf{b} \wedge \mathsf{b}_I)$$

Here kb is set of other knowledge-base elements, conf contains the agent's internal representation, $x_1, y_1, v_1$ are fresh symbols and $\mathsf{b}_I$ is set of constraints: $x_1 = (x + (v + v_1) \times \mathsf{dt}/2) \wedge y_1 = y \wedge v_1 = v + \mathsf{acc} \times \mathsf{dt}$. These constraints specify the agent's new position and speed using classical physics equations.

The verification property bad where an agent is too close to an obstacle, such as a pedestrian, is specified by the search command:

$$\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow$$
$$([\mathsf{env} \mid \mathsf{at}(\mathsf{ag}_1, (x_1, y_1)), \mathsf{at}(\mathsf{ag}_2, (x_2, y_2)), \mathsf{kb}]\ \mathsf{conf}, \mathsf{b}')\ \mathsf{such\ that}\ \mathsf{SMT}(\mathsf{b}' \wedge \mathsf{b}_I)$$

where $\mathsf{b}_I$ is the set of constraints: $x_1 = x_2 \wedge y_1 = y_2$, specifying that two agents $\mathsf{ag}_1$ and $\mathsf{ag}_2$ are in the same position, i.e., colliding.

## 3 Hybrid BFS-DFS Algorithm

The definition of Incremental Rewrite Theories addresses the problem of the **Definition of Increments** discussed above. The second problem (**Not possible to chain incremental calls**) still needs to be addressed. Indeed, BFS procedures do not enable the chaining of incremental calls. To illustrate this, consider again the search tree and BFS execution in Figure 1. Assume that $\mathsf{b}_1 = \mathsf{b}_0 \wedge \mathsf{b}_{0,1}, \mathsf{b}_2 = \mathsf{b}_0 \wedge \mathsf{b}_{0,2}$ and that $\mathsf{goalCond}(\mathsf{t}, \mathsf{b})$ has the form $\mathsf{b} \wedge \mathsf{b}_I$ as one would expect when using Incremental Rewrite Theories. It is possible to call SMT incrementally during the sequence of calls $\mathsf{SMT}(\mathsf{b}_1)$ and $\mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_1, \mathsf{b}_1))$, but not chain incrementally the call $\mathsf{SMT}(\mathsf{b}_2)$. This is because it is not possible to define an increment between $\mathsf{b}_1$ and $\mathsf{b}_2$ as they lie in different branches of the search tree.

The first obvious alternative is using Depth-First Search (DFS) instead of BFS. This would indeed lead to an execution that could chain incremental calls to the SMT. For example, in the tree depicted in Figure 1, the sequence of calls would be

$$\mathsf{SMT}(\mathsf{b}_0); \mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_0, \mathsf{b}_0)); \mathsf{SMT}(\mathsf{b}_1); \mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_1, \mathsf{b}_1)); \\ \mathsf{SMT}(\mathsf{b}_3); \mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_3, \mathsf{b}_3)) \ldots$$

Since $\mathsf{b}_3$ is of the form $\mathsf{b}_0 \wedge \mathsf{b}_{0,1} \wedge \mathsf{b}_{1,3}$, we know the increment is $\mathsf{b}_{1,3}$. There are, however, two problems with DFS. The first problem is that DFS may not find a solution that could be found using BFS due to an infinite branch. The second problem is that the sequence of call using $\mathsf{goalCond}(\mathsf{t}, \mathsf{b})$ appears in between the increments, e.g., $\mathsf{SMT}(\mathsf{b}_0); \mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_0, \mathsf{b}_0)); \mathsf{SMT}(\mathsf{b}_1)$.

We propose the algorithm hybrid_search described in Figure 2 that addresses these two problems of DFS by combining BFS and DFS and using the PUSH and POP features of SMT-solvers for incremental solving. These features enable the creation of backtracking scopes of learned clauses. By default, sequential calls to SMT will attempt to use incremental solving based on the constraints solved in previous calls. A call to PUSH will add to the solver stack any learned clauses from calls to SMT while a call to POP will remove any learned clauses since the last PUSH.

The hybrid_search algorithm takes as input the search tree $T$[6], a non-negative natural number $d$, and a goal condition $g$. Intuitively, the parameter $d$ specifies the depth to which the algorithm shall perform DFS before switching to BFS.

We start with *Queue* empty and a *Solver*. hybrid_search starts at line 4 with the next few lines initializing $found$ to be NULL and pushing the root of $T$ onto *Queue*. The while loop starts with line 7 continuing while *Queue* is non empty and no solution has been found. It pops the next node off the *Queue* on line 8, then calls dfs_bounded on the next line using this node as the root starting on line 12. dfs_bounded is a modified depth-bounded depth-first search. It starts with creating a backtracking scope on *Solver* by calling PUSH and storing the result $\mathsf{SMT}(b)$ where $b$ is the boolean constraint of the current node.

[6] Notice that in practice, there is a mechanism that constructs the tree on the fly.

```
1: Queue : FIFO Queue
2: Solver : SMT Solver
3: found : Node
4: function hybrid_search(tree, depth, goal)
5:     found ← NULL
6:     push root of tree on Queue
7:     while Queue has elements and found is NULL do
8:         node ← Queue.pop()
9:         dfs_bounded(node, depth, goal, 0)
10:    end while
11: end function
12: function dfs_bounded(node, max_depth, goal, curr_depth)
13:    b ← node.getBoolean()
14:    Solver.push()
15:    rsat ← Solver.check(b)
16:    if rsat is UNSAT then
17:        Solver.pop() return
18:    end if
19:    if goal(node) then
20:        found ← node
21:        return
22:    end if
23:    if curr_depth = max_depth then
24:        for all child ∈ node.children() do
25:            Queue.add(child)
26:        end for
27:        return
28:    end if
29:    for all child ∈ node.children() do
30:        dfs_bounded(child, max_depth, goal, curr_depth + 1)
31:        Solver.pop()
32:    end for
33: end function
```

**Fig. 2.** Pseudo-code of the Hybrid Search Algorithm hybrid_search.

Subsequently, in line 16, it checks if SMT($b$) returned UNSAT, and if so, we POP and return immediately and not explore any children of this node. Any descendent nodes would have a boolean constraint of the form $b \wedge b_I$ for some $b_I$, and since SMT($b$) is UNSAT it must be the case that $b \wedge b_I$ is also UNSAT. Otherwise, we continue with checking if $goal(node)$ is true on line 19 and if so setting $found$ to this node and then terminating dfs_bounded and hybrid_search. If $found$ is not set, then line 23 checks when the current depth is equal to the depth parameter $d$ and if it is we add all of the children nodes, i.e.all the nodes that are $d+1$ depth away from the initial root node called from line 9, to $Queue$ and no more nodes at a lower depth are visited for now. After all such nodes are added, the execution returns to line 7 to start another dfs_bounded from the next element in $Queue$. Until then, it continues traversing the tree in a DFS-

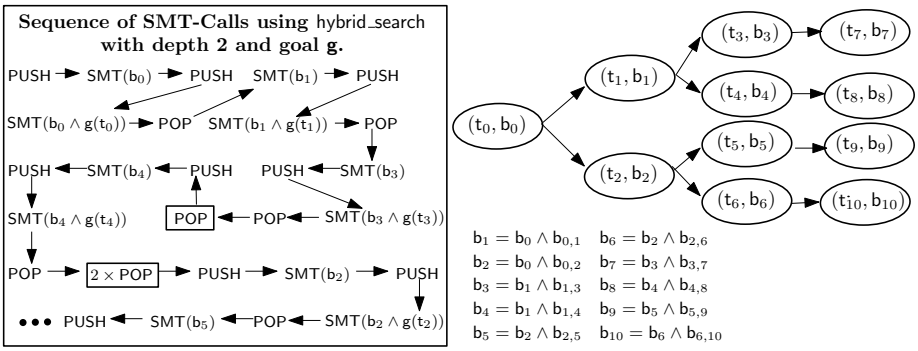


**Fig. 3.** Illustration of an hybrid_search algorithm execution using the goal condition g and depth two. The POP surrounded by a box indicates the points when the algorithm back-tracks in the search tree.

like manner on line 29 ensuring that when dfs_bounded backtracks, we call POP for each node, and hence it backtracks such that *Solver* can properly unlearn clauses that it no longer needs.

We illustrate the execution of hybrid_search with the tree shown in Figure 3. It also contains the sequence of calls to PUSH, POP and SMT due to the initial call to dfs_bounded. The sequence of calls illustrates the chaining of incremental calls to the SMT. For example, the data-structures constructed in the call $\mathsf{SMT}(\mathsf{b}_1)$ are used in the SMT calls for $\mathsf{b}_3, \mathsf{b}_4$, including the calls $\mathsf{goal}(\mathsf{b}_3)$ and $\mathsf{goal}(\mathsf{b}_4)$. This makes sense as $\mathsf{b}_1$ is sub-formula of $\mathsf{b}_3$, $\mathsf{b}_4$, $\mathsf{goal}(\mathsf{b}_3)$ and $\mathsf{goal}(\mathsf{b}_4)$. However, the data-structures constructed in the SMT call for $\mathsf{goal}(\mathsf{b}_1)$ is not stored due to the subsequent POP call, as $\mathsf{goal}(\mathsf{b}_1)$ is not necessarily a subformula of $\mathsf{b}_3$, $\mathsf{b}_4$, $\mathsf{goal}(\mathsf{b}_3)$ and $\mathsf{goal}(\mathsf{b}_4)$. The second observation is the combination of DFS and BFS. When the subtree of depth $d = 2$ is traversed, the algorithm removes the data-structures constructed during the call of $\mathsf{SMT}(\mathsf{b}_1)$, indicated by the $2 \times \mathsf{POP}$ in Figure 3, as $\mathsf{b}_1$ is not necessarily a subformula of $\mathsf{b}_2$.

Notice that the depth parameter ($d$) plays the role of specifying how much incremental solving one is willing to use with the risk of traversing longer a branch of the search tree that may not have a solution. For example, in the tree and execution shown in Figure 3, the algorithm will traverse the node $(\mathsf{t}_7, \mathsf{b}_7)$ and will call $\mathsf{SMT}(\mathsf{b}_7)$, but without using the data-structures constructed previously for $\mathsf{b}_3$, that is, it will not solve it incrementally.

The following results relate hybrid_search with BFS and with DFS.

**Proposition 1.** *Let $T$ be a tree and $g$ be a decidable goal condition. Then,* $\mathsf{hybrid\_search}(T, 0, g)$ *will traverse $T$ in the same order as BFS.*

**Proposition 2.** *Let $T$ be a tree and $g$ be a decidable goal condition. Suppose the depth of $T$ is $d$. Then, for any $k \geq d$,* $\mathsf{hybrid\_search}(T, k, g)$ *will traverse $T$ in the same order as DFS.*

The following statement provides coverage guarantees.

**Proposition 3.** *Let $d > 0$, $T$ be a tree of finite branching, and $g$ be a decidable goal condition. Then,* hybrid_search$(T, d, g)$ *finds a solution in finite time, i.e.some node $n$ in $T$ such that $g(n)$ is true, if such a solution exists.*

**Proof.** Let $B_i$ be the number of nodes in $T$ at depth $i$. Suppose that the solution node $n$ exists at depth $r$ and no solutions exist at a lower depth. Let $0 \leq r \leq qd$ for some $q$. The first depth-bounded DFS will traverse all nodes up to depth $d$. This then adds $B_{d+1}$ nodes to *Queue*. Running the depth-bounded DFS run these nodes will traverse all the nodes to $2d$. Traversing all nodes up to $qd$ would take $1 + B_{d+1} + B_{d+2} + ... + B_{qd}$ iterations of depth-bounded depth first searches. Since $n$ exists at depth $r \leq qd$ and each $B_i$ is finite since $T$ has finite branching, $n$ would be found in finite time. **QED**.

To address the fact that search trees may have infinite depth, often one uses bounded search that searches the tree until only some given depth $d$. The following proposition states that in these cases it is best to deploy hybrid_search with depth $d$ to search through all nodes of the sub-tree, provided incremental SMT calls are more efficient than SMT calls from scratch.

**Proposition 4.** *Let $T$ be a tree of finite branching with branching factor $b$ and $g$ be a decidable goal condition. Let $T(d)$ be the sub-tree of $T$ of depth $d$ with $d > 0$. Assume that incremental SMT calls, i.e., using* PUSH*, take less time than calls from scratch, i.e., without using* PUSH*. Then for any $d' \geq 0$ such that $d' \neq d$, the time required by* hybrid_search$(T, d, g)$ *to traverse all nodes in $T(d)$ is less than the time of* hybrid_search$(T, d', g)$ *to traverse all nodes in $T(d)$.*

**Proof.** Let $0 < r < 1$ be the average performance benefit from incremental SMT calls and $t$ be the time it takes for non-incremental SMT calls. Let $B_i$ be the number of nodes at depth $i$. Since $b$ is finite, each $B_i$ is finite. The time required by hybrid_search$(T, d, g)$ to traverse all nodes in $T(d)$ is $t + rtB_1 + rtB_2 + ... + rtB_d$. Suppose that $0 < d' < d$. Let $pd' < d \leq (p+1)d'$ for some $p$. For hybrid_search$(T, d', g)$ to traverse all nodes in $T(d)$, it must traverse all nodes in $T((p+1)d')$ because each dfs_bounded must travel exactly $d'$ depth, hybrid_search$(T, d', g)$ will traverse only depths that are multiples of $d'$. Then, the time required for hybrid_search$(T, d', g)$ is $t+rtB_1+...+rtB_{d'}+tB_{d'+1}+rtB_{d'+2}+$ $...rtB_{2d'} + ... + tB_{pd'} + rtB_{pd'+1} + ... + rtB_{(p+1)d'}$. There are $p+1$ terms that do not get the benefit from incremental SMT calls for hybrid_search$(T, d', g)$ while there is 1 term that does not get this benefit for hybrid_search$(T, d, g)$. Hence, the time required for hybrid_search$(T, d, g)$ to traverse all nodes in $T(d)$ is less than the time required for hybrid_search$(T, d', g)$ to traverse all nodes in $T(d)$. Now, suppose that $d' > d$. Then, for hybrid_search$(T, d', g)$ to traverse all nodes in $T(d)$, it must traverse all nodes in $T(d')$. The time required for hybrid_search$(T, d', g)$ is $t + rtB_1 + rtB_2 + ... + rtB_{d'}$. But, because $d' > d$ and each $rtB_i > 0$ the time required for hybrid_search$(T, d, g)$ is less than hybrid_search$(T, d', g)$. Hence, the time required for hybrid_search$(T, d, g)$ to traverse all nodes in $T(d)$ is less than the time required for hybrid_search$(T, d', g)$ to traverse all nodes in $T(d)$. Therefore, for any $d' \neq d$ the the time required for hybrid_search$(T, d, g)$ to


**Fig. 4.** Overview of the implementation used for the experiments using hybrid_search, the SMT solver Z3 and the rewriting tool Maude.

traverse all nodes in $T(d)$ is less than the time required for hybrid_search$(T, d', g)$ to traverse all nodes in $T(d)$. **QED**.

## 4 Implementation and Experiments

Our implementation is based on Python with the Z3 SMT solver and Maude integrated using Python bindings [15] as depicted in Figure 4. The Z3 Solver is responsible for checking the incremental satisfiability of constraints using SMT, PUSH and POP, while Maude is responsible for executing rewriting rules. The Maude bindings allow for loading Maude files into the Python implementation of hybrid_search. The search is done with a Python function that repeatedly calls the Maude search with one step (Search1) so that the traversal of the search space can be controlled. The original Maude specifications were modified to replace calls to SMT with calls to functions defined using the Maude hook mechanism for attaching external code to function symbols. This mechanism is exposed by the Maude Python bindings. There are two types of function, one that checks satisfiability while keeping any learned clauses from the check, and one that just checks without adding any learned clauses. The functions keep track of the SMT solver state using appropriate calls to PUSH and POP. The implementation is available at [17].

Figures 5, 6 and 7 summarize the experiments carried out using implementations available in the literature [3,16,13] for the verification of the systems described in Examples 1, 2, and 3. All experiments were run on a Windows 10 machine, Intel Core i7-10700J, 16 GB of RAM, on Python 3.10.2, using Maude python bindings 1.1.2 and Z3 4.11.2.0. We measure the runtime for these three applications of rewriting modulo SMT to determine the performance gain from using hybrid search at various depth parameters compared to BFS and DFS. Each table shows the initial configuration for the system, then statistics for searches for BFS, DFS, and using hybrid_search at various depths terminating when finding a single goal node. The statistics have the form $n/m/p$ which specify the time $n$ in seconds to perform verification, the number of states $m$ traversed, and the percentage $p$ of verification time required by SMT-solving. DNF indicates that no solution was found within 30 minutes. For example, the first row for cashOK$_1$ using the BFS mechanism for instance, the execution time was 6.9 seconds, requiring 91 state traversals while spending 77% of execution time in Z3.

For our experiments, we used the same subsets of the verification problems used in references [3,16,13]:

| Init | BFS | DFS | HYBRID d=2 | HYBRID d=4 | HYBRID d=8 |
|---|---|---|---|---|---|
| $\mathsf{cashOK}_1$ | 6.9 / 93 / 77% | 0.7 / 9 / 8% | **0.2 / 37 / 21%** | 1.3 / 117 / 12% | 0.7 / 9 / 8% |
| $\mathsf{cashOK}_2$ | 8.0 / 100 / 71% | 3.0 / 12 / 4% | **0.6 / 52 / 13%** | 1.9 / 118 / 9.0% | 0.9 / 14 / 6% |
| $\mathsf{cashOK}_3$ | 3.5 / 65 / 84% | DNF | 0.1 / 32 / 25% | **0.06 / 9 / 26%** | 1.3 / 28 / 6% |
| $\mathsf{cashBad}_1$ | 5.9 / 63 / 74% | 0.7 / 9 / 7% | **0.2 / 27 / 21%** | 1.2 / 61 / 10.% | 0.7 / 9 / 8% |
| $\mathsf{cashBad}_2$ | 7.5 / 70 / 69% | 2.9 / 12 / 4% | **0.6 / 42 / 13%** | 1.8 / 62 / 7.7% | 0.9 / 14 / 6% |
| $\mathsf{cashBad}_3$ | 2.6 / 39 / 81% | DNF | **0.1 / 22 / 26%** | 0.06 / 9 / 24% | 1.4 / 28 / 6% |

**Fig. 5.** CASH Verification Experiments. $\mathsf{cashOK}_1 = \mathsf{cashOK}(I0, I1, I2, I3, true)$, $\mathsf{cashOK}_2 = \mathsf{cashOK}(I0, I1, I2, I3, I0 + I3 > I1 + I2)$, and $\mathsf{caseOK}_3 = \mathsf{caseOK}(I0, I1, I2, I1, I0 + I2 > I1)$, and *mutatis mutandis* for $\mathsf{cashBad}_1$, $\mathsf{cashBad}_2$ and $\mathsf{cashBad}_3$.

| Init | BFS | DFS | HYBRID d=2 | HYBRID d=3 | HYBRID d=4 |
|---|---|---|---|---|---|
| $\mathsf{Slow}_1$ | 2.4 / 51 / 88% | 0.2 / 66 / 39% | 0.3 / 52 / 56% | 0.4 / 79 / 57% | **0.2 / 35 / 41%** |
| $\mathsf{Slow}_2$ | 39.8 / 775 / 83% | DNF | 8.0 / 1612 / 44% | **3.1 / 703 / 39%** | 13.0 / 3314 / 36% |
| $\mathsf{Slow}_3$ | 0.5 / 11 / 87% | 0.06 / 10 / 50% | 0.06 / 9 / 46% | **0.05 / 8 / 52%** | 0.06 / 9 / 50% |
| $\mathsf{Slow}_4$ | 1.8 / 29 / 86% | 0.1 / 27 / 39% | 0.2 / 27 / 55% | 0.2 / 34 / 54% | **0.1 / 20 / 41%** |
| $\mathsf{Slow}_5$ | 19.0 / 147 / 78% | DNF | 2.5 / 187 / 44% | 1.3 / 118 / 41% | 3.9 / 261 / 39% |

**Fig. 6.** Slowloris Experiments. $\mathsf{Slow}_1 = \mathsf{Slowloris}(1, 0, 24)$, $\mathsf{Slow}_2 = \mathsf{Slowloris}(1, 0, 36)$, $\mathsf{Slow}_3 = \mathsf{Slowloris}(1, 1, 12)$, $\mathsf{Slow}_4 = \mathsf{Slowloris}(1, 1, 24)$, $\mathsf{Slow}_5 = \mathsf{Slowloris}(1, 1, 36)$.

| Init | BFS | DFS | HYBRID $d = t$ | HYBRID $d = 2 \times t$ | HYBRID $d = 3 \times t$ |
|---|---|---|---|---|---|
| $\mathsf{cps}_1$ | 12.3 / 119 / 62% | **4.8 / 57 / 68%** | 7.0 / 117 / 65% | 6.6 / 99 / 67% | 5.0 / 57 / 71% |
| $\mathsf{cps}_2$ | 53.4 / 323 / 71% | 23.0 / 152 / 78% | **15.6 / 213 / 69%** | 28.6 / 232 / 77% | 22.7 / 152 / 78% |
| $\mathsf{cps}_3$ | 301.0 / 819 / 84% | 97.3 / 387 / 85% | 63.9 / 429 / 80% | **52.7 / 364 / 82%** | 99.7 / 387 / 85% |
| $\mathsf{cps}_4$ | 12.5 / 119 / 63% | 4.8 / 57 / 70% | 7.8 / 118 / 69% | 6.4 / 99 / 66% | **4.7 / 57 / 68%** |
| $\mathsf{cps}_5$ | 56.0 / 323 / 72% | 25.4 / 152 / 80% | **18.4 / 227 / 71%** | 19.8 / 192 / 74% | 23.3 / 152 / 79% |
| $\mathsf{cps}_6$ | 285.1 / 819 / 83% | 100.9 / 387 / 85% | **60.2 / 424 / 79%** | 84.6 / 437 / 85% | 101.4 / 387 / 85% |

**Fig. 7.** Cyber-Physical System Verification Experiments, where $\mathsf{cps}_1 = \mathsf{pedestrian}(3, 3, 2, 1)$, $\mathsf{cps}_2 = \mathsf{pedestrian}(4, 3, 2, 1)$, $\mathsf{cps}_3 = \mathsf{pedestrian}(5, 3, 2, 1)$, $\mathsf{cps}_4 = \mathsf{pedestrian}(3, 4, 2, 1)$, $\mathsf{cps}_5 = \mathsf{pedestrian}(4, 4, 2, 1)$, $\mathsf{cps}_6 = \mathsf{pedestrian}(5, 4, 2, 1)$. The bound $t$, $2 \times t$ and $3 \times t$ is determined according to the $t$ parameter of the scenario.

– $\mathsf{cashOK}(I_0, I_1, I_2, I_3, \mathsf{b})$ and $\mathsf{cashBad}(I_0, I_1, I_2, I_3, \mathsf{b})$ correspond to symbolic initial configurations of a CASH scheduling problem with two servers (see Example 1). $I_0$ and $I_1$ specify, respectively, the maximum budget and the period of the first server, while $I_2$ and $I_3$ specify, respectively, the maximum budget and period of the second server. $\mathsf{b}$ is a constraint on the values of $I_1, I_2, I_3$, and $I_4$. $\mathsf{cashOK}$ uses a correct implementation of the scheduler, while $\mathsf{cashBad}$ uses an incorrect specification.

– Slowloris($P_1, P_2$, DoSDur) corresponds to symbolic initial configurations of a Slowloris verification problem (see Example 2). $P_1$ specifies the bound on the number of parallel bursts of symbolic protocols, and $P_2$ specifies the bound on the number of different types of messages sent in parallel, where $P_2 = 0$ denotes no bound. Moreover, DoSDur specifies the minimum duration for which the server's resources are depleted in order to consider the DoS attack successful.
– pedestrian($t$, Safer, Safe, Unsafe) specifies a pedestrian crossing scenario problem where an autonomous vehicle is approaching a pedestrian crossing. The verification problem is to avoid an unsafe situation. The three levels of safety are defined according to the parameters Safer > Safe > Unsafe specifying bounds on the distance to between the vehicle and the pedestrian measured in terms of time to travel. The verification problem is to determine whether a given vehicle controller cannot reach an unsafe situation within $t$ time units when starting at a safe situation. The size of a time unit is $0.1s$

The results for the CASH verification experiments show that hybrid_search finishes up to about 10 times faster than BFS and terminates in all cases as opposed to two of the DFS cases where it does not finish within 30 minutes. The overhead of Z3 is reduced from about 70% to 80% down to 6% to 25% from BFS to hybrid_search. This indicates the effectiveness of the incremental SMT solving for the types of constraints used in this example.

Similarly, in the Slowloris examples, hybrid_search finishes up to 10 times faster than BFS with termination while two of the DFS cases do not finish within 30 minutes. In these cases the overhead of Z3 goes from about 80% to 90% in BFS while it goes from about 30% to 60% in hybrid_search, demonstrating the effectiveness of the incremental solving. Interestingly, even when there is a much larger number of states traversed, e.g., in case $\mathsf{Slow}_2$ and HYBRID d=4 with 3314 states traversed as opposed to 775 states traversed by BFS, the verification time is one third, from 40s to 13s. This indicates that main overhead of BFS is indeed SMT solving.

For the Cyber-Physical System (CPS) Verification experiments, hybrid_search completes up to about 5 times faster than BFS. The overhead of Z3 does not change significantly in these experiments, which indicates that the incremental solving is not as effective as in the other two examples (CASH and Slowloris). The reason for this may be the non-linear nature of the constraints for CPS systems which contrast with the former two examples that use linear arithmetic constraints. Despite this, hybrid_search and DFS still outperform BFS because they need to traverse less nodes before finding a goal node.

## 5 Related Work

We consider three related areas of work in optimizing symbolic execution modulo SMT, hybrid search strategies, incremental constraint solving methods, and tradeoffs between search space and constraint complexity.

*Hybrid search strategies.* There have been others that have previously explored techniques of combining BFS and DFS so to take advantage of both of their benefits while reducing the drawbacks of each.

Reference [5] proposes a hybrid algorithm for Binary Decision Diagrams (BDDs). BDDs are are often used to represent and manipulate boolean functions symbolically. Traditionally, depth-first approaches were used in the construction of BDDs as it had relatively low memory overhead. Though, it had been discovered that using a breadth-first approach instead had better performance due to better memory access locality at the cost of larger memory overhead. To improve upon both approaches a hybrid of the two is used. Essentially, the algorithm switches between the two techniques based on its memory overhead. When the memory overhead is computed to be low, a breadth-first search is used and when it is high a depth-first search is used.

Reference [1] constructs a "breadth-first, depth-next" algorithm for building Random Forest (RF) models. An RF model is a machine learning model that uses decision trees. Both DFS and BFS approaches are used in machine learning frameworks. They observe that BFS has memory efficient access patterns at lower depths. As the depth increases it loses this benefit and virtually has random access to memory. At this point, DFS performs better. As a result, their algorithm starts with a breadth-first approach until it is computed that is no longer has efficient access pattern, switching to a depth-first approach.

Reference [9] introduces "depth-first iterative-deepening (DFID)." One of the issues with BFS is that it has exponential memory complexity. DFS can circumvent this drawback as its memory complexity is linear, but comes with its own problems. It generally requires some depth bound and check for repeated nodes, otherwise the search may not terminate. The actual depth bound needed may not be knowable at runtime and choosing a bound too low may result in the search ending without finding the solution. To counteract the downsides of BFS and DFS, DFID is used. DFID starts with DFS bounded by depth one, then performs a DFS bounded by depth two, and continue this process with incrementally larger bounded depths until a solution is found. It must visit the same nodes multiple times, but it is shown that the runtime complexity is not effected by it.

Unfortunately, none of these algorithms seem particularly helpful with respect to rewriting modulo SMT. For example, prior algorithms [5,1] attempt to take advantage of memory locality as much as possible. In our case, it would not give us much performance increase. Reference [9] requires nodes to be visited multiple times. This would lead to duplicate calls the SMT solver, only increasing the bottleneck.

*Incremental solving.* In reference [10] the authors compare cache-based and stack-based incremental constraint solving methods in the context of symbolic execution for test generation. Cached-based incrementality works outside the solver to cache results and attempt to reuse them. Stack-based incrementality uses a solvers ability to reuse information learned when solving a subproblem and the associated push/pop interface. Implementations of the two methods and a

baseline (no incrementality) were compare on large benchmark set of C programs and on randomly generated programs. The space of symbolic execution paths was searched using bounded depth first search. The authors found that caching generally increased average solving time over baseline (by a factor of 2-5 depending on code size), while stack-based methods decreased average solving time by roughly a factor of 20. This is consistent with our observations even though the source of search tree is different and the class of constraints is different.

*Trading search space for constraint complexity.* A notion of guarded term is introduced in reference [2] as a method to reduce the search state space in symbolic rewriting modulo SMT by replacing non-determinism by disjunction. The effect of using guarded terms is demonstrated in a study of the CASH algorithm for task scheduling. Many properties that could not be checked using symbolic execution modulo SMT (due to size of search space and timeout) became tractable using guarded terms.

A study of the tradeoff between search space size and constraint size using symbolic execution modulo SMT in the context of analyzing safety of autonomous systems such platooning scenarios is presented in reference [13]. The results in that present paper suggest that not only the size of state space matters for automation, but also the size of constraints that are sent to the SMT-Solver as many searches fail to terminate due to non-termination of constraint solving when constraints get large, while the same searches terminate with disjunctions are turned into branching in the search space.

None of these approaches, however, investigate the use of incremental SMT solving for improving performance of Rewriting Modulo SMT.

## 6 Conclusions and Future Work

This paper proposes Incremental Rewrite Theories that enable incremental SMT solving for rewriting modulo SMT. This is accomplished by the search procedure hybrid_search which combines BFS and DFS. The effectiveness of hybrid_search is demonstrated by using a collection of verification problems taken from the literature, including algorithm verification, network security analysis, and cyber-physical systems safety verification. In all examples, the time taken to verify by hybrid_search improved by a factor between 5-10 when compared to traditional BFS approaches, showing the great benefits of using incremental solving. As future work, we are investigating the trade-offs of incremental solving and the shape of constraints, e.g., use disjunctions to reduce search space versus split disjunctions to reduce SMT solving time. We also are investigating the incorporation of incremental solving algorithms in tool implementations such as Maude.

## References


1. A. Anghel, N. Ioannou, T. Parnell, N. Papandreou, and C. Mendler-Dünner. Breadth-first, depth-next training of random forests. In *Neural Information Processing Systems (NeurIPS)*, 2019.
2. K. Bae and C. Rocha. Symbolic state space reduction with guarded terms for rewriting modulo SMT. *Sci. Comput. Program.*, 178:20–42, 2019.
3. K. Bae and C. Rocha. Symbolic state space reduction with guarded terms for rewriting modulo SMT. In *Formal Aspects of Component Software (FACS)*, 2019.
4. M. Caccamo, G. C. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), Orlando, Florida, USA, 27-30 November 2000*, pages 295–304. IEEE Computer Society, 2000.
5. Y. Chen, B. Yang, and R. Bryant. Breadth-first with depth-first BDD construction: A hybrid approach, 1997.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
7. Y. G. Dantas, V. Nigam, and I. E. Fonseca. A selective defense for application layer ddos attacks. In *IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014, The Hague, The Netherlands, 24-26 September, 2014*, pages 75–82. IEEE, 2014.
8. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *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*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
9. R. Korf. Depth-first iterative-deepening: An optimal admissible tree search, 1985.
10. T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri. A comparative study of incremental constraint solving approaches in symbolic execution. In E. Yahav, editor, *Hardware and Software: Verification and Testing*, pages 284–299, Cham, 2014. Springer International Publishing.
11. MaudeSE. https://github.com/maude-se/maude-se.github.io. 2021.
12. J. Meseguer. Conditional Rewriting Logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
13. V. Nigam and C. Talcott. Automating safety proofs about cyber-physical systems using rewriting modulo smt. In K. Bae, editor, *14th International Workshop on Rewriting Logic and its Applications*, volume 13252 of *LNCS*, pages 212–229. Springer, 2022.
14. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT and open system analysis. *Journal of Logical and Algebraic Methods in Programming*, pages 269–297, 2017.
15. R. Rubio. Maude as a library: an efficient all-purpose programming interface. In *Rewriting Logic and its Applications (WRLA)*, 2022.
16. A. A. Urquiza, M. A. Alturki, T. B. Kirigin, M. I. Kanovich, V. Nigam, A. Scedrov, and C. L. Talcott. Resource and timing aspects of security protocols. *J. Comput. Secur.*, 29(3):299–340, 2021.
17. G. Whitters, B. Loo, V. Nigam, and C. Talcott. Incremental rewriting modulo smt experiments. https://github.com/WhittersGerald/cade-incremental-rewriting, 2023.