Speculative SAT Modulo SAT

Hari Govind V K
University of Waterloo
Waterloo, Canada
hgvedira@uwaterloo.ca

Isabel Garcia-Contreras
University of Waterloo
Waterloo, Canada
igarciac@uwaterloo.ca

Sharon Shoham
Tel-Aviv University
Tel-Aviv, Israel
sharon.shoham@cs.tau.ac.il

Arie Gurfinkel
University of Waterloo
Waterloo, Canada
arie.gurfinkel@uwaterloo.ca

***Abstract*—State-of-the-art model-checking algorithms like IC3/PDR are based on uni-directional modular SAT solving for finding and/or blocking counterexamples. Modular SAT-solvers divide a SAT-query into multiple sub-queries, each solved by a separate SAT-solver (called a module), and propagate information (lemmas, proof obligations, blocked clauses, etc.) between modules. While modular solving is key to IC3/PDR, it is obviously not as effective as monolithic solving, especially when individual sub-queries are harder to solve than the combined query. This is partially addressed in SAT modulo SAT (SMS) by propagating unit literals back and forth between the modules and using information from one module to simplify the sub-query in another module as soon as possible (i.e., before the satisfiability of any sub-query is established). However, bi-directionality of SMS is limited because of the strict order between decisions and propagation – only one module is allowed to make decisions, until its sub-query is SAT. In this paper, we propose a generalization of SMS, called SPECSMS, that *speculates* decisions between modules. This makes it bi-directional – decisions are made in multiple modules, and learned clauses are exchanged in both directions. We further extend DRUP proofs and interpolation, these are useful in model checking, to SPECSMS. We have implemented SPECSMS in Z3 and show that it performs exponentially better on a series of benchmarks that are provably hard for SMS.**

I. Introduction

IC3/PDR [3] is an efficient SAT-based Model Checking algorithm. Among many other innovations in IC3/PDR is the concept of a modular SAT-solver that divides a formula into multiple *frames* and each frame is solved by an individual SAT solver. The solvers communicate by exchanging proof obligations (i.e., satisfying assignments) and lemmas (i.e., learned clauses).

While modular reasoning in IC3/PDR is very efficient for a Model Checker, it is not as efficient as a classical monolithic SAT-solver. This is not surprising since modularity restricts the solver to colorable refutations [11], which are, in the worst case, exponentially bigger than unrestricted refutations. On the positive side, IC3/PDR's modular SAT-solving makes interpolation trivial, and enables generalizations of proof obligations and inductive generalization of lemmas – both are key to the success of IC3/PDR.

This motivates the study of modular SAT-solving, initiated by SMS [1]. Our strategic vision is that our study will contribute to improvements in IC3/PDR. However, in this paper, we focus on modular SAT-solving in isolation.

In modular SAT-solving, multiple solvers interact to check satisfiability of a partitioned CNF formula, where each part of the formula is solved by one of the solvers. In this paper, for simplicity, we consider the case of two solvers $\langle S_\mathsf{s}, S_\mathsf{m}\rangle$ checking satisfiability of a formula pair $\langle \Phi_\mathsf{s}, \Phi_\mathsf{m}\rangle$. S_m is a *main* solver and S_s is a *secondary* solver. In the notation, the solvers are written right-to-left to align with IC3/PDR, where the main solver is used for frame 1 and the secondary solver is used for frame 0.

When viewed as a modular SAT-solver, IC3/PDR is uni-directional. First, S_m finds a satisfying assignment σ to Φ_m and only then, S_s extends σ to an assignment for Φ_s. Learned clauses, called *lemmas* in IC3/PDR, are only shared (or copied) from the secondary solver S_s to the main solver S_m.

SAT Modulo SAT (SMS) [1] is a modular SAT-solver that extends IC3/PDR by allowing inter-modular unit propagation and conflict analysis: whenever an interface literal is placed on a trail of any solver, it is shared with the other solver and both solvers run unit propagation, exchanging unit literals. This makes modular SAT-solving in SMS bi-directional as information flows in both directions between the solvers. Bi-directional reasoning can simplify proofs, but it significantly complicates conflict analysis. To manage conflict analysis, SMS does not allow the secondary solver S_s to make any decisions before the main solver S_m is able to find a complete assignment to its clauses. As a result, learned clauses are either local to each solver, or flow only from S_s to S_m, restricting the structure of refutations similarly to IC3/PDR.

Both IC3/PDR and SMS require S_m to find a complete satisfying assignment to Φ_m before the solving is continued in S_s. This is problematic since Φ_m might be hard to satisfy, causing them to get stuck in Φ_m, even if considering both formulas together quickly reveals the (un)satisfiability of $\langle \Phi_\mathsf{s}, \Phi_\mathsf{m}\rangle$.

In this paper, we introduce SPECSMS — a modular SAT-solver that employs a truly bi-directional reasoning. SPECSMS builds on SMS, while facilitating deeper communication between the modules by (1) allowing learnt clauses to flow in both directions, and (2) letting the two solvers interleave their decisions. The key challenge is in the adaptation of conflict analysis to properly handle the case of a conflict that depends on decisions over local variables of both solvers. Such a conflict cannot be explained to either one of the solvers using only interface clauses (i.e., clauses over interface variables). It may, therefore, require backtracking the search without learning any conflict clauses. To address this challenge, SPECSMS uses *speculation*, which tames decisions of the secondary solver that are interleaved with decisions of the main solver. If the secondary solver satisfies all of its clauses during speculation, a *validation* phase is employed, where the main solver attempts to extend the assignment to satisfy its unassigned clauses. If speculation leads to a conflict which depends on local decisions of both solvers, *refinement* is employed to resolve the conflict. Refinement ensures progress even if no conflict clause can be learnt. With these ingredients, we show that SPECSMS is sound and complete (i.e., always terminates).

To certify SPECSMS's result when it determines that a formula is unsatisfiabile, we extract a *modular* clausal proof from its execution. To this end, we extend DRUP proofs [12] to account for modular reasoning, and devise a procedure for trimming modular proofs. Such proofs are applicable both to SPECSMS and to SMS. Finally, we propose an interpolation algorithm that extracts an interpolant [4] from a modular proof. Since clauses are propagated between the solvers in both directions, the extracted interpolants have the shape $\bigwedge_i (C_i \Rightarrow cls_i)$, where C_i are conjunctions of clauses and each cls_i is a clause.

Original SMS is implemented on top of MiniSAT. For this paper, we implemented both SMS and SPECSMS in Z3 [5], using the extendable SAT-solver interface of Z3. Thanks to its bi-directional reasoning, SPECSMS is able to efficiently solve both sat and unsat formulas that are provably hard for existing modular SAT-solvers, provided that speculation is performed at the right time. SPECSMS relies on a user-provided callback to decide when to speculate. Developing good heuristics for speculation is a topic of future work.

In summary, we make the following contributions: (i) the SPECSMS algorithm that leverages bi-directional modular reasoning (Sec. III); (ii) modular DRUP proofs for SPECSMS (Sec. IV-A); (iii) proof-based interpolation algorithm; (iv) user interface to guide speculation and search (Sec. V); and (v) implementation and validation (Sec. VI).

II. MOTIVATING EXAMPLES

In this section, we discuss two examples in which both IC3/PDR-style uni-directional reasoning and SMS-style shallow bi-directional reasoning are ineffective. The examples illustrate why existing modular reasoning gets stuck. To better convey our intuition, we present our problems at word level using bit-vector variables directly, without explicitly converting them to propositional variables.

Example 1 Consider the following modular sat query: $\langle \varphi_{in}, \varphi_{\text{SHA-1}} \rangle$, where $\varphi_{in} \triangleq (in = in_1) \vee (in = in_2)$, in is a 512-bit vector, in_1, in_2 are 512-bit values, $\varphi_{\text{SHA-1}} \triangleq (\text{SHA-1}_{circ}(in) = \text{SHA-1}_{in_1})$, $\text{SHA-1}_{circ}(in)$ is a circuit that computes SHA-1 of *in*, and SHA-1_{in_1} is the 20 byte SHA-1 message digest of in_1.

Checking the satisfiability of $\varphi_{in} \wedge \varphi_{\text{SHA-1}}$ is easy because it contains both the output and the input of the SHA-1 circuit. However, existing modular SAT-solvers attempt to solve the problem starting by finding a complete satisfying assignment to $\varphi_{\text{SHA-1}}$. This is essentially the problem of inverting the SHA-1 function, which is known to be very hard for a SAT-solver. The improvements in SMS do not help. There are no unit clauses in φ_{in}.

On the other hand, SPECSMS proceeds as follows: (1) when checking satisfiability of $\varphi_{\text{SHA-1}}$, it decides to speculate, (2) it starts checking satisfiability of φ_{in}, branches on variables in, finds an assignment σ to in and unit propagates σ to $\varphi_{\text{SHA-1}}$, (3) if there is a conflict in $\varphi_{\text{SHA-1}}$, it learns the conflict clause $in \neq in_2$, and (4) it terminates with a satisfying assignment $in = in_1$. Speculation in step (1) is what differentiates SPECSMS from IC3/PDR and SMS. The specifics of when exactly SPECSMS speculates is guided by the user, as explained in Sec. V. □

Example 2 Speculation is desirable for unsatisfiable formulas as well. Consider the modular sat query $\langle \varphi_+, \varphi_- \rangle$, where $\varphi_+ \triangleq (a < 0 \Rightarrow x) \wedge (a \geq 0 \Rightarrow x) \wedge PHP^1_{32}$ and $\varphi_- \triangleq (b < 0 \Rightarrow \neg x) \wedge (b \geq 0 \Rightarrow \neg x) \wedge PHP^2_{32}$. Here, a and b are 32-wide bitvectors and local to the respective modules. PHP_{32} encodes the problem of fitting 32 pigeons into 31 holes and PHP^1_{32} and PHP^2_{32} denote a partitioning of PHP_{32} into 2 problems such that both formulas contain all variables. The modular problem $\langle \varphi_+, \varphi_- \rangle$ is unsatisfiable, x and PHP^1_{32} being two possible interpolants. IC3/PDR and SMS only find the second interpolant. This is because, all satisfying assignments to φ_- immediately produce a conflict in PHP^1_{32} part of φ_+. However, learning an interpolant containing x requires searching (i.e., deciding) in both φ_+ and φ_-. SPECSMS, with proper guidance, solves the problem by (1) deciding on all b variables in φ_-, (2) switching to speculation, (3) branching on all a variables in φ_+ to hit a conflict in x, and, finally (4) learning the conflict clause x. □

These examples highlight the need to speculate while doing modular reasoning. While speculation by itself is quite powerful, it requires proper guidance to be effective. We explain how the user can provide such a guidance in Sec. V.

III. SPECULATIVE SAT MODULO SAT

This section presents SPECSMS — a modular bi-directional SAT algorithm. For simplicity, we restrict our attention to the case of two modules. However, the algorithm easily generalizes to any sequence of modules.

A. Sat Modulo Sat

We assume that the reader has some familiarity with internals of a MiniSAT-like SAT solver [6] and with SMS [1]. We give a brief background on SMS, highlighting some of the key aspects. SMS decides satisfiability of a partitioned CNF formula $\langle \Phi_{\mathsf{s}}, \Phi_{\mathsf{m}} \rangle$ with a set of shared interface variables I. It uses two modules $\langle S_{\mathsf{s}}, S_{\mathsf{m}} \rangle$, where S_{m} is a *main* module used to solve Φ_{m}, and S_{s} is a *secondary* module to solve Φ_{s}. Each module is a SAT solver (with a slightly extended interface, as described in this section). We refer to them as *modules* or *solvers*, interchangeably. Each solver has its own clause database (initialized with Φ_i for $i \in \{\mathsf{m}, \mathsf{s}\}$), and a trail of literals, just as a regular SAT solver. The solvers keep their decision levels in sync. Whenever a decision is made in one solver, the decision level of the other solver is incremented as well (adding a *null* literal to its trail if necessary). Whenever one solver back-jumps to level i, the other solver back-jumps to level i as well. Assignments to interface variables are shared between the solvers: whenever such a literal is added to the trail of one solver (either as a decision or due to propagation), it is also added to the trail of the other solver. SMS requires that S_{s} does not make any decisions, until S_{m} finds a satisfying assignment to its clauses.

Inter-modular propagation and conflict analysis: The two key features of SMS are inter-modular unit propagation (called PropagateAll in [1]) and the corresponding inter-modular conflict analysis. In PropagateAll, whenever an interface literal is added to the trail of one solver, it is added to the trail of the other, and both solvers run unit propagation. Whenever a unit literal ℓ is copied from the trail of one solver to the other, the `reason` for ℓ in the destination solver is marked using a marker ext. This indicates that the justification for the unit is external to the destination solver[1]. Propagation continues until either there are no more units to propagate or one of the solvers hits a conflict.

Conflict analysis in SMS is extended to account for units with no reason clauses. If such a literal ℓ is used in conflict analysis, its reason is obtained by using $\mathsf{AnalyzeFinal}(\ell)$ on the other solver to compute a clause $(s \Rightarrow \ell)$ over the interface literals. This clause is copied to the requesting solver and is used as the missing reason. Multiple such clauses can be copied (or learned) during analysis of a single conflict clause – one clause for each literal in the conflict that is assigned by the other solver.

In SMS, it is crucial that $\mathsf{AnalyzeFinal}(\ell)$ always succeeds to generate a reason clause over the interface variables. This is ensured by only calling $\mathsf{AnalyzeFinal}(\ell)$ in the S_{s} solver on literals that were added to the trail when S_{s} was not yet making decisions. This can happen in one of two scenarios: either S_{m} hits a conflict due to literals propagated from S_{s}, in which case $\mathsf{AnalyzeFinal}$ is invoked in S_{s} on each literal marked ext in S_{m} that is involved in the conflict resolution to obtain its `reason`; or S_{s} hits a conflict during unit propagation, in which case it invokes $\mathsf{AnalyzeFinal}$ to obtain a conflict clause over the interface variables that blocks the partial assignment of S_{m}. In both cases, new reason clauses are always copied from S_{s} to S_{m}. We refer the reader to [1] for the pseudo-code of the above inter-modular procedures for details.

B. Speculative Sat Modulo Sat

specSMS extends SMS [1] by a combination of *speculation*, *refinement*, and *validation*. During the search in the main solver S_{m}, specSMS non-deterministically *speculates* by allowing the secondary solver S_{s} to extend the current partial assignment of Φ_{m} to a satisfying assignment of Φ_{s}. If S_{s} is unsuccessful (i.e., hits a conflict), and the conflict depends on a combination of a *local* decision of S_{m} with some decision of S_{s}, then the search reverts to S_{m} and its partial assignment is *refined* by forcing S_{m} to decide on an interface literal from the conflict. On the other hand, if S_{s} is successful, solving switches to the main solver S_{m} that *validates* the current partial assignment by extending it to all of its clauses. This either succeeds (meaning, $\langle \Phi_{\mathsf{s}}, \Phi_{\mathsf{m}} \rangle$ is sat), or fails and another *refinement* is initiated. Note that the two subcases where S_{s} is unsuccessful but the reason for the conflict is either local to S_{s} or local to S_{m} are handled as in SMS.

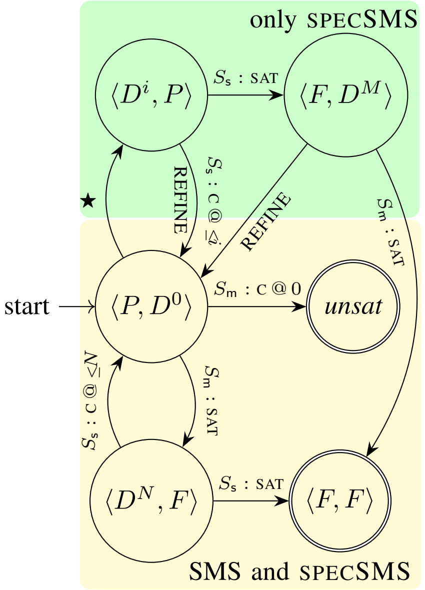

Fig. 1: State transitions of specSMS. A state $\langle P, D^0\rangle$ means that the secondary solver S_{s} is in propagate mode and the main solver S_{m} is in solve mode. Each edge is guarded with a condition. The condition S_{m} : SAT means that S_{m} found a full satisfying assignment to Φ_{m}. The condition S_{m} : C @ $\trianglelefteq j$ means that S_{m} hit a conflict at a decision level below j. The four states in yellow corresponds to SMS; two states in green are unique to specSMS.

Search modes: specSMS controls the behavior of the solvers and their interaction through *search modes*. Each solver can be in one of the following search modes: Decide, Propagate, and Finished. In Decide, written D^i, the solver treats all decisions below level i as assumptions and is allowed to both make decisions and do unit propagation. In Propagate, written P, the solver makes no decisions, but does unit propagation whenever new literals are added to its trail. In Finished, written F, the clause database of the solver is satisfied; the solver neither makes decisions nor propagates unit literals.

The pair of search modes of both modules is called the *state* of specSMS, where we add a unique state called *unsat* for the case when the combination of the modules is known to be unsatisfiable. The possible states and transitions of specSMS are shown in Fig. 1. States *unsat* and $\langle F, F\rangle$ are two final states, corresponding to unsat and sat, respectively. In all other states, exactly one of the solvers is in a state D^i. We refer to this solver as *active*. The part of the transition system highlighted in yellow correspond to SMS, and the green part includes the states and transitions that are unique to specSMS.

Normal execution with bi-directional propagation: specSMS starts in the state $\langle P, D^0\rangle$, with the main solver being

[1] This is similar to theory propagation in SMT solvers.

active. In this state, it can proceed like SMS by staying in the yellow region of Fig. 1. We call this *normal execution with bi-directional propagation*, since (only) unit propagation goes between solvers.

Speculation: What sets SPECSMS apart is speculation: at any non-deterministically chosen decision level i, SPECSMS can pause deciding on the main solver and activate the secondary solver (i.e., transition to state $\langle D^i, P\rangle$). During speculation, only the secondary solver makes decisions. Since the main solver does not have a full satisfying assignment to its clauses, the secondary solver propagates assignments to the main solver and vice-versa.

Speculation terminates when the secondary solver S_s either: (1) hits a conflict that cannot be resolved by inter-modular conflict analysis; (2) hits a conflict below decision level i; or (3) finds a satisfying assignment to Φ_s.

Case (1) is most interesting, and is what makes SPECSMS differ from SMS. Note that a conflict clause is not resolved by inter-modular conflict analysis only if it depends on an external literal on the trail of S_s that cannot be explained by an interface clause from S_m. This is possible when both S_m and S_s have partial assignments during speculation. So the conflict might depend on the *local* decisions of S_m. This cannot be communicated to S_s using only interface variables.

Refinement: In SPECSMS, this is handled by modifying the REASON method in the solvers to fail (i.e., return ext) whenever AnalyzeFinal returns a non-interface clause. Additionally, the literal on which AnalyzeFinal failed is recorded in a global variable *refineLit*. This is shown in Alg. 1. The inter-modular conflict analysis is modified to exit early whenever REASON fails to produce a justification. At this point, SPECSMS exits speculation, returns to the initial state $\langle P, D^0\rangle$, both solvers back-jump to decision level i at which speculation was initiated, and S_m is forced to decide on *refineLit*.

Algorithm 1 The REASON method in modular SAT solvers inside SPECSMS

```
function REASON($lit$)
    if $reason[lit] = \text{ext}$ then
        $c \leftarrow other.\mathsf{AnalyzeFinal}(lit)$
        if $\exists v \in c \cdot v \notin I$ then
            $refineLit \leftarrow lit$
            return ext
        ADDCLAUSE($c$)
        $reason[lit] \leftarrow c$
    return $reason[lit]$
```

We call this transition a *refinement* because the partial assignment of the main solver S_m (which we view as an *abstraction*) is updated (a.k.a., refined) based on the information that was not available to it (namely, a conflict with a set of decisions in the secondary solver S_s). Since *refineLit* was not decided on in S_m prior to speculation, deciding on it is a new decision that ensures progress in S_m. The next speculation is possible only under strictly more decisions in S_m than before, or when S_m back-jumps and flips an earlier decision.

We illustrate the refinement process on a simple example:

Example 3 Consider the query $\langle\Phi_\mathsf{s}, \Phi_\mathsf{m}\rangle$ with:

$\Phi_\mathsf{s}(i, j, k, z)$:

$$\overline{z} \lor \overline{i} \qquad (3)$$

$$i \lor j \lor \overline{k} \qquad (4)$$

$\Phi_\mathsf{m}(a, i, j, k)$:

$$\overline{a} \lor i \lor \overline{j} \qquad (1)$$

$$j \lor k \qquad (2)$$

First, S_m decides a (at level 1), which causes no propagations. Then, SPECSMS enters speculative mode, transitions to $\langle D^1, P\rangle$ and starts making decisions in S_s. S_s decides z and calls PROPAGATEALL. Afterwards, the trails for S_m and S_s are as follows:

S_m	a @ 1	*null* @ 2	$\overline{i}$ (ext)	$\overline{j}$ (1)	k (2)
S_s	*null* @ 1	z @ 2	$\overline{i}$ (3)	$\overline{j}$ (ext)	k (ext)

where x @ i denotes that literal x is decided at level i, and x (r) denotes that literal x is propagated using a reason clause r, or due to the other solver (if $r = \text{ext}$). A conflict is hit in S_s in clause (4). Inter-modular conflict analysis begins. S_s first asks for the reason for k, which is clause (2) in S_m. This clause is copied to S_s. Note that unlike SMS, clauses can move from S_m to S_s. The new conflict to be analyzed is $(i \lor j \lor j)$. Now the reason for $\overline{j}$ is asked of S_m. In this case, S_m cannot produce a clause over shared variables to justify $\overline{j}$, so conflict analysis fails with *refineLit* $= j$. This causes SPECSMS to exit speculation mode and move to state $\langle P, D^0\rangle$ and S_m must decide variable j before speculating again. Note that in this case either decision on j results in $\langle\Phi_\mathsf{s}, \Phi_\mathsf{m}\rangle$ being sat. ■

Case (2) is similar to what happens in SMS when a conflict is detected in S_s. The reason for the conflict is below level i which is below the level of any decision of S_s. Since decision levels below i are treated as assumptions in S_s, calling AnalyzeFinal in S_s returns an interface clause c that blocks the current assignment in S_m. The clause c is added to S_m. The solvers back-jump to the smallest decision level j that makes c an asserting clause in S_m. Finally, SPECSMS moves to $\langle P, D^0\rangle$.

Validation: Case (3), like Case (1), is unique to SPECSMS. While all clauses of S_s are satisfied, the current assignment might not satisfy all clauses of S_m. Thus, SPECSMS enters *validation* by switching to the configuration $\langle F, D^M\rangle$, where M is the current decision level. Thus, S_m becomes active and starts deciding and propagating. This continues, until one of two things happen: (3a) S_m extends the assignment to satisfy all of its clauses, or (3b) a conflict that cannot be resolved with inter-modular conflict analysis is found. In the case (3a), SPECSMS transitions to $\langle F, F\rangle$ and declares that $\langle\Phi_\mathsf{m}, \Phi_\mathsf{s}\rangle$ is sat. The case (3b) is handled exactly the same as Case (1) – the literal on the trail without a reason is stored in *refineLit*, SPECSMS moves to $\langle P, D^0\rangle$, backjumps to the level in which speculation was started, and S_m is forced to decide on *refineLit*.

Theorem 1 SPECSMS *terminates. If it reaches the state* $\langle F, F\rangle$*, then* $\Phi_\mathsf{s} \land \Phi_\mathsf{m}$ *is satisfiable and the join of the trails of* $\langle S_\mathsf{s}, S_\mathsf{m}\rangle$ *is a satisfying assignment. If it reaches the state unsat,* $\Phi_\mathsf{s} \land \Phi_\mathsf{m}$ *is unsatisfiable.* □

IV. Validation and interpolation

In this section, we augment specSMS with an interpolation procedure. To this end, we first introduce modular DRUP proofs, which are generated from specSMS in a natural way. We then present an algorithm for extracting an interpolant from a modular trimmed DRUP proof in the spirit of [11].

A. DRUP proofs for modular SAT

Modular DRUP proofs – a form of clausal proofs [9] – extend (monolithic) DRUP proofs [12]. A DRUP proof [12] is a sequences of steps, where each step either asserts a clause, deletes a clause, or adds a new Reverse Unit Propagation (RUP) clause. Given a set of clauses Γ, a clause cls is an RUP for Γ, written $\Gamma \vdash_{UP} cls$, if cls follows from Γ by unit propagation [8]. For a DRUP proof π, let $\textsc{asserted}(\pi)$ denote all clauses of the asserted commands in π, then π shows that all RUP clauses of π follow from $\textsc{asserted}(\pi)$. If π contains a $\bot$ clause, then π certifies $\textsc{asserted}(\pi)$ is unsat.

A Modular DRUP proof is a sequence of clause addition and deletion steps, annotated with indices idx (m or s). Intuitively, steps with the same index must be validated together (within the same module idx), and steps with different indices may be checked independently. The steps are:

1) $(\text{asserted}, idx, cls)$ denotes that cls is asserted in idx,
2) (rup, idx, cls) denotes adding RUP clause cls to idx,
3) $(\text{cp}(src), dst, cls)$ denotes copying a clause cls from src to dst, and
4) (del, idx, cls) denotes removing clause cls from idx.

We denote the prefix of length k of a sequence of steps π by π^k. Given a sequence of steps π and a formula index idx, we use $act\_clauses(\pi, idx)$ to denote the set of active clauses with index idx. Formally,

$$\begin{aligned}&\{cls \mid \exists c_j \in \pi \cdot\\ &(c_j = (t, idx, cls) \wedge (t = \text{asserted} \vee t = \text{rup} \vee t = \text{cp}(\_)))\\ &\qquad\qquad \wedge \ \ \neg\exists c_k \in \pi \cdot k > j \wedge c_k = (\text{del}, idx, cls)\}\end{aligned}$$

A sequence of steps $\pi = c_1, \ldots, c_n$ is a *valid modular DRUP proof* iff for each $c_i \in \pi$:

1) if $c_i = (\text{rup}, idx, cls)$ then $act\_clauses(\pi^i, idx) \vdash_{UP} cls$,
2) if $c_i = (\text{cp}(idx), \_, cls)$ then $act\_clauses(\pi^i, idx) \vdash_{UP} cls$, and
3) $c_{|\pi|}$ is either $(\text{rup}, \mathsf{m}, \bot)$ or $(\text{cp}(\mathsf{s}), \mathsf{m}, \bot)$.

Let $\textsc{asserted}(\pi, idx)$ be the set of all asserted clauses in π with index idx.

Theorem 2 *If π is a valid modular DRUP proof, then $\textsc{asserted}(\pi, \mathsf{s}) \wedge \textsc{asserted}(\pi, \mathsf{m})$ is unsatisfiable.* □

Modular DRUP proofs may be validated with either one or two solvers. To validate with one solver we convert the modular proof into a monolithic one (i.e., where the steps are asserted, rup, and del). Let modDRUP2DRUP be a procedure that given a modular DRUP proof π, returns a DRUP proof π' that is obtained from π by (a) removing idx from all the steps; (b) removing all cp steps; (c) removing all del steps. Note that del steps are removed for simplicity, otherwise it is necessary to account for deletion of copied and non-copied clauses separately.

Algorithm 2 Trimming a modular DRUP proof

Input: Solver instances S_s, S_m with the empty clause on the trail, and a modular clausal proof $\pi = c_1, \ldots, c_n$.
Output: A proof π' s.t. all steps are core.

```
$\pi' = \emptyset$
$M_\mathsf{s}, M_\mathsf{m} \leftarrow \{\bot\}, \emptyset$                    ▷ Relevant clauses
for $i = n$ to $0$ do
    match $c_i$ with $(type, idx, cls)$
    if $cls \notin M_{idx}$ then continue
    if $type = \text{del}$ then
        $S_{idx}$.Revive($cls$)
        continue
    $\pi'.append(c_i)$
    if $type = \text{rup}$ then
        $S_{idx}$.CHK_RUP($cls, M_{idx}$)
    else if $type = \text{cp}(src)$ then
        $S_{idx}$.Delete($cls$)
        $M_{src}.add(cls)$
$\pi'.reverse()$
function SOLVER::CHK_RUP($cls, M$)
    if IsOnTrail($cls$) then
        UndoTrail($cls$)
    Delete($cls$)
    SaveTrail()
    Enqueue($\neg cls$)
    $r \leftarrow$ Propagate()
    ConflictAnalysis($r, M$)          ▷ Updates $M$ with conflict clauses
    RestoreTrail()
```

Lemma 1 *If π is a valid modular DRUP proof then $\pi' =$ modDRUP2DRUP(π) is a valid DRUP proof.* □

Modular validation is done with two monolithic solvers working in lock step: $(\text{asserted}, cls, idx)$ steps are added to the idx solver; (rup, idx, cls) steps are validated locally in solver idx using all active clauses (asserted, copied, and rup); and for $(\text{cp}(src), dst, cls)$ steps, cls is added to dst but not validated in it, and cls is checked to exist in the src solver.

From now on, we consider only valid proofs. We say that a (valid) modular DRUP proof π is a proof of unsatisfiability of $\Phi_\mathsf{s} \wedge \Phi_\mathsf{m}$ if $\textsc{asserted}(\pi, \mathsf{s}) \subseteq \Phi_\mathsf{s}$ and $\textsc{asserted}(\pi, \mathsf{m}) \subseteq \Phi_\mathsf{m}$ (inclusion here refers to the sets of clauses).

specSMS produces modular DRUP proofs by logging the clauses that are learnt, deleted, and copied between solvers. Note that in SMS clauses may only be copied from S_s to S_m, but in specSMS they might be copied in both directions.

Theorem 3 *Let Φ_s and Φ_m be two Boolean formulas s.t. $\Phi_\mathsf{s} \wedge \Phi_\mathsf{m} \models \bot$. specSMS produces a valid modular DRUP proof for unsatisfiability of $\Phi_\mathsf{s} \wedge \Phi_\mathsf{m}$.* □

Trimming modular DRUP proofs. A step in a modular DRUP proof π is *core* if removing it invalidates π. Under this definition, del steps are never core since removing them does not affect validation. Alg. 2 shows an algorithm to trim modular DRUP proofs based on backward validation. The input are two modular solvers S_m and S_s in a final conflicting state, and a valid modular DRUP proof $\pi = c_1, \ldots, c_n$. The output is a trimmed proof π' such that all steps of π' are core.

We assume that the reader is familiar with MiniSAT [6] and use the following solver methods: Propagate, exhaustively applies unit propagation (UP) rule by resolving all unit clauses; ConflictAnalysis analyzes the most recent conflict and marks which clauses are involved in the conflict; IsOnTrail checks whether a clause is an antecedent of a literal on the trail; Enqueue enqueues one or more literals on the trail; IsDeleted, Delete, Revive check whether a clause is deleted, delete a clause, and add a previously deleted clause, respectively; SaveTrail, RestoreTrail save and restore the state of the trail.

Alg. 2 processes the steps of the proof backwards, rolling back the states of the solvers. M_{idx} marks which clauses were relevant to derive clauses in the current suffix of the proof. While the proof is constructed through inter-modular reasoning, the trimming algorithm processes each of the steps in the proof completely locally. During the backward construction of the trimmed proof, steps that include unmarked clauses are ignored (and, in particular, not added to the proof). For each (relevant) rup step, function CHK_RUP, using ConflictAnalysis, adds clauses to M. del steps are never added to the trimmed proof, but the clause is revived from the solver. For cp steps, if the clause was marked, it is marked as used for the solver it was copied from and the step is added to the proof. Finally, asserted clauses that were marked are added to the trimmed proof. Note that, as in [11], proofs may be trimmed in different ways, depending on the strategy for ConflictAnalysis.

The following theorem states that trimming preserves validity of the proof.

Theorem 4 *Let Φ_{s} and Φ_{m} be two formulas such that $\Phi_{\mathsf{s}} \wedge \Phi_{\mathsf{m}} \models \bot$. If π is a modular DRUP proof produced by solvers S_{s} and Φ_{m} for $\Phi_{\mathsf{s}} \wedge \Phi_{\mathsf{m}}$, then a trimmed proof π' by Alg. 2 is also a valid modular DRUP proof for $\Phi_{\mathsf{s}} \wedge \Phi_{\mathsf{m}}$.* □

Fig. 2 shows a trimmed proof after SPECSMS is executed on $\langle \psi_0, \psi_1 \rangle$ such that $\psi_0 \triangleq ((s_1 \wedge la_1) \Rightarrow s_2)) \wedge ((s_1 \wedge \neg la_1) \Rightarrow s_2) \wedge ((s_3 \wedge la_2) \Rightarrow s_4) \wedge ((s_3 \wedge \neg la_2) \Rightarrow s_4)$ and $\psi_1 \triangleq (\neg s_1 \Rightarrow lb_1) \wedge (\neg s_1 \Rightarrow \neg lb_1) \wedge ((s_2 \wedge lb_2) \Rightarrow s_3) \wedge ((s_2 \wedge \neg lb_2) \Rightarrow s_3) \wedge (s_4 \Rightarrow lb_3) \wedge (s_4 \Rightarrow \neg lb_3))$.

B. Interpolation

Given a modular DRUP proof π of unsatisfiability of $\Phi_{\mathsf{s}} \wedge \Phi_{\mathsf{m}}$, we give an algorithm to compute an interpolant of $\Phi_{\mathsf{s}} \wedge \Phi_{\mathsf{m}}$. For simplicity of the presentation, we assume that π has no deletion steps; this is the case in trimmed proofs, but we can also adapt the interpolation algorithm to handle deletions by keeping track of active clauses.

Our interpolation algorithm relies only on the clauses copied between the modules. Notice that whenever a clause is copied from module i to module j, it is implied by all the clauses in Φ_i together with all the clauses that have been copied from module j. We refer to clauses copied from S_{m} to S_{s} as *backward* clauses and clauses copied from S_{s} to S_{m} as *forward* clauses. The conjunction of forward clauses is unsatisfiable with S_{m}. This is because, in the last step of π, $\bot$ is added to S_{m}, either through rup or by cp $\bot$ from S_{s}. Since all the clauses in module m are implied by Φ_{m} together with forward clauses, this means that the conjunction of forward clauses is unsatisfiable with Φ_{m}. In addition, all forward clauses were learned in module s, with support from backward clauses. This means that every forward clause is implied by Φ_{s} together with the subset of the backward clauses used to derive it. Intuitively, we should therefore be able to learn an interpolant with the structure: backward clauses imply forward clauses.

seq	step	to	clause
1	asserted	m	$\neg s_1 \Rightarrow lb_1$
2	asserted	m	$\neg s_1 \Rightarrow \neg lb_1$
3	asserted	s	$(s_1 \wedge la_1) \Rightarrow s_2$
4	asserted	s	$(s_1 \wedge \neg la_1) \Rightarrow s_2$
5	asserted	m	$(s_2 \wedge lb_2) \Rightarrow s_3$
6	asserted	m	$(s_2 \wedge \neg lb_2) \Rightarrow s_3$
7	asserted	s	$(s_3 \wedge la_2) \Rightarrow s_4$
8	asserted	s	$(s_3 \wedge \neg la_2) \Rightarrow s_4$
9	asserted	m	$s_4 \Rightarrow lb_3$
10	asserted	m	$s_4 \Rightarrow \neg lb_3$
11	rup	m	s_1
12	rup	m	$\neg s_4$
13	rup	m	$s_2 \Rightarrow s_3$
14	cp(m)	s	$s_2 \Rightarrow s_3$
15	rup	s	$s_3 \Rightarrow s_4$
16	rup	s	$s_1 \Rightarrow s_4$
17	cp(s)	m	$s_1 \Rightarrow s_4$
18	rup	m	$\bot$

Fig. 2: An example of a modular DRUP proof. Clauses are written in human-readable form as implications, instead of in the DIMACS format.

Algorithm 3 Interpolating a modular DRUP proof.

```
Input: Propositional formulas $\langle \Phi_0, \Phi_1 \rangle$
Input: A modular trimmed DRUP proof $\pi = c_1, \ldots, c_n$ of unsatisfiability of $\Phi_0 \wedge \Phi_1$
Output: An interpolant $itp$ s.t. $\Phi_0 \Rightarrow itp$ and $itp \wedge \Phi_1 \models \bot$
$S_{\mathsf{s}}, S_{\mathsf{m}} \leftarrow$ SAT_SOLVER()
$itp \leftarrow \top$
for $i = 0$ to $n$ do
    match $c_i$
        with (asserted, s, $cls$):
            $sup(cls) \leftarrow \top$
        with (cp(m), s, $cls$):
            $sup(cls) \leftarrow cls$
        with (rup, s, $cls$):
            $M \leftarrow \emptyset$
            $S_{\mathsf{s}}$.CHK_RUP($cls, M$)
            $sup(cls) \leftarrow \{sup(c) \mid c \in M\}$
        with (cp(s), m, $cls$):
            $itp \leftarrow itp \wedge (sup(cls) \Rightarrow cls)$
    $S_{c_i.idx}.add(cls)$
```

Alg. 3 describes our interpolation algorithm. It traverses a modular DRUP proof forward. For each clause cls learned in module s, the algorithm collects the set of backward clauses used to learn cls. This is stored in the sup datastucture — a mapping from clauses to sets of clauses. Finally, when a forward clause c is copied, it adds $\sup(c) \Rightarrow c$ to the interpolant.

Example 4 We illustrate our algorithm using the modular DRUP proof from Fig. 2. On the first cp step (cp(m), s, $s_2 \Rightarrow s_3$), the algorithm assigns the sup for clause $s_2 \Rightarrow s_3$ as itself (line 8). The first clause learnt in module s, $(\mathsf{rup}, \mathsf{s}, s_3 \Rightarrow s_4)$, is derived from just the clauses in module s and no backward clauses. Therefore, after RUP, our algorithm sets $\sup(s_3 \Rightarrow s_4)$ to $\top$ (line 12). The second clause learnt in module s, $s_1 \Rightarrow s_4$, is derived from module s with the support of the backward clause $s_2 \Rightarrow s_3$. Therefore, $\sup(s_1 \Rightarrow s_4) = \{s_2 \Rightarrow s_3\}$. When this clause is copied forward to module 1, the algorithm updates the interpolant to be $(s_2 \Rightarrow s_3) \Rightarrow (s_1 \Rightarrow s_4)$. ■

Next, we formalize the correctness of the algorithm. Let $L_B(\pi) = \{cls \mid (\mathsf{cp}(\mathsf{m}), \mathsf{s}, cls) \in \pi\}$ be the set of clauses copied from module m to s and $L_F(\pi) = \{cls \mid (\mathsf{cp}(\mathsf{s}), \mathsf{m}, cls) \in \pi\}$ be clauses copied from module s to m. From the validity of modular DRUP proofs, we have that:

Lemma 2 *For any step $c_i = (cp(\mathsf{s}), \mathsf{m}, cls) \in \pi$, $(L_B(\pi^i) \wedge \Phi_{\mathsf{s}}) \Rightarrow cls$ and for any step $c_j = (cp(\mathsf{m}), \mathsf{s}, cls) \in \pi$, $(L_F(\pi^j) \wedge \Phi_{\mathsf{m}}) \Rightarrow cls$.* □

For any clause cls copied from one module to the other, we use the shorthand $\sharp(cls)$ to refer to the position of the copy command in the proof π. That is, $\sharp(cls)$ is the smallest k such that $c_k = (\mathsf{cp}(i), j, cls) \in \pi$. The following is an invariant in a valid modular DRUP proof:

Lemma 3

$$\forall cls \in L_F(\pi) \cdot (\Phi_{\mathsf{m}} \wedge (L_F(\pi^{\sharp(cls)})) \Rightarrow L_B(\pi^{\sharp(cls)})) \quad \square$$

These properties ensure that adding $L_B(\pi^{\sharp(cls)}) \Rightarrow cls$ for every forward clause cls results in an interpolant. Alg. 3 adds $(\sup(cls) \Rightarrow cls)$ as an optimization. Correctness is preserved since $\sup(cls)$ is a subset of $L_B(\pi^{\sharp(cls)})$ that together with Φ_{s} suffices to derive cls (formally, $\sup(cls) \wedge \Phi_{\mathsf{s}} \vdash_{UP} cls$).

Theorem 5 *Given a modular DRUP proof π for $\Phi_{\mathsf{s}} \wedge \Phi_{\mathsf{m}}$, $itp \triangleq \{\sup(c) \Rightarrow c \mid c \in L_F(\pi)\}$ is an interpolant for $\langle \Phi_{\mathsf{s}}, \Phi_{\mathsf{m}} \rangle$.* □

PROOF Since all copy steps are over interface variables, the interpolant is also over interface variables. By Lemma 2 (and the soundness of sup optimization), $\Phi_{\mathsf{s}} \Rightarrow itp$. Next, we prove that $(\Phi_{\mathsf{m}} \wedge itp) \Rightarrow \bot$. From Lemma 3, we have that for all $c \in L_F(\pi)$, $(\Phi_{\mathsf{m}} \wedge L_F(\pi^{\sharp(c)})) \Rightarrow \sup(c)$. Therefore, $(\Phi_{\mathsf{m}} \wedge L_F(\pi^{\sharp(c)}) \wedge (\sup(c) \Rightarrow c)) \Rightarrow c$ ■

It is much simpler to extract interpolants from modular DRUP proofs then from arbitrary DRUP proofs. This is not surprising since the interpolants capture exactly the information that is exchanged between solvers. The interpolants are not in CNF, but can be converted to CNF after extraction.

V. GUIDING SPECSMS VIA SOLVER CALLBACKS

As the reader may have noticed, deciding when to switch to speculative mode is non-trivial. Heuristics may be implemented, as typically done in SAT solvers, but we consider this out of the scope of this paper. Instead, we provide an interface for users to guide SPECSMS based on solver callbacks. This scheme has been recently proven useful to guide SMT solving and to define custom theories in Z3 [2].

Users may provide a function NextSplit to guide the solver in whether to speculate and over which variables to decide. SPECSMS calls NextSplit whenever the next decision is about to be made. SPECSMS expects NextSplit to return $none$ (default, in which case, the underlying heuristics are used) or a pair $(ch\_mode, Vars)$ where ch_mode is a Boolean that indicates a change to speculative mode and $Vars$ is a (possibly empty) set of variables to assign. To implement NextSplit, users can ask the solver if a variable has been assigned using the IsFixed function. We illustrate the API with some examples.

Example 5 Consider modular queries of the following form: $\langle \psi_{in}(\ell, in), \psi_{\text{SHA-1}}(in, out) \rangle$, where ℓ is a 2-bit vector, in is a 512-bit vector (shared), out is 160-bit vector. ψ_{in} encodes that there are four possible messages:

$$\begin{aligned}\psi_{in} \triangleq (\ell = 0 \wedge in = msg_0) \vee (\ell = 1 \wedge in = msg_1) \vee \\ (\ell = 2 \wedge in = msg_2) \vee (\ell = 3 \wedge in = msg_3)\end{aligned}$$

and $\psi_{\text{SHA-1}}(in, out)$ encodes the SHA-1 circuit together with some hash:

$$\psi_{\text{SHA-1}} \triangleq (\text{SHA-1}_{circ}(in) \wedge out = shaVal)$$

Roughly, the modular query $\langle \psi_{in}, \psi_{\text{SHA-1}} \rangle$ asks whether the SHA-1 of any msg_i is $shaVal$. As we saw in Sec. II, we are interested in using speculation in queries of this form in order to avoid the hard problem of inverting SHA-1 (as required by SMS, for example). We can guide the solver with the function $\mathsf{NextSplit}() \triangleq (\top, \ell)$. ■

Speculation is useful for such queries both in cases where the formulas are satisfiable and unsatisfiable. If unsat, only the 4 inputs for SHA-1 specified by msg_i need to be considered, avoiding the expensive hash inversion problem. If sat, only two decisions on the bits of ℓ result in fully assigning in, which results again in just checking the hash.

Example 6 Next, consider a different form of modular query: $\langle \gamma_{in}(\ell, x, in), \gamma_{\text{SHA-1}}(in, x, out) \rangle$, where x is an 512-bit vector, ℓ is a 160-bit vector, $chks_i$ are 512-bit vector, and the remaining variables are the same as in ψ_{in} and $\psi_{\text{SHA-1}}$, and

$$\begin{aligned}\gamma_{in} \triangleq\ & \text{SHA-1}_{circ}(x, \ell) \wedge \\ & \big((\ell = chks_0 \wedge in = msg_0) \vee (\ell = chks_1 \wedge in = msg_1) \vee \\ & (\ell = chks_2 \wedge in = msg_2) \vee (\ell = chks_3 \wedge in = msg_3)\big) \\ \gamma_{\text{SHA-1}} \triangleq\ & (x = 1 \vee x = 4) \wedge \text{SHA-1}_{circ}(in, out) \wedge out = shaVal\end{aligned}$$

This is an example where bi-directional search is necessary to efficiently solve the query. If deciding only on $\gamma_{\text{SHA-1}}$, we encounter the hard problem of inverting SHA-1_{circ}, if speculating directly in γ_{in}, we encounter the same problem, since an assignment for x needs to be found, based on the four values for ℓ.

Accordingly, in this case, we are not interested in speculating immediately, but rather first decide on the value of x in

	time (s) – sat			time (s) – unsat	
# rounds	SMS	SPECSMS	# rounds	SMS	SPECSMS
16	1.96	0.41	16	0.77	0.65
21	–	0.66	21	–	0.89
26	–	0.66	26	–	0.91
31	–	0.81	31	–	1.08
36	–	1.01	36	–	1.45
40	–	1.16	40	–	1.77

TABLE I: Solving time with a timeout of 600s.

the main solver and then speculate. The following NextSplit implementation captures this idea:

$$\begin{aligned} \mathsf{NextSplit}() \triangleq\ & \textbf{if}(\textbf{not}\ \mathsf{IsFixed}(x))\ (\bot, x) \\ & \textbf{else if}(\textbf{not}\ \mathsf{IsFixed}(\ell))\ (\top, \ell) \\ & \textbf{else}\ \textit{none} \end{aligned} \qquad \blacksquare$$

This example gives an intuition on which instances SPECSMS is better than SMS. Even if SMS is guided by NextSplit, at least one inversion of SHA-1$_{circ}$ would have to be computed.

VI. Implementation and Validation

We have implemented SPECSMS (and SMS) inside the extensible SAT-solver of Z3 [5]. For SMS, we simply disable speculation. The code is publicly available on GitHub[2].

We have validated SPECSMS on a set of handcrafted benchmarks, based on Ex. 5, using the query $\langle \psi_{in}, \psi_{\text{SHA-1}} \rangle$. In the first set of experiments, we check sat queries by generating one msg_i in ψ_{in} that matches $shaVal$. In the second set, we check unsat queries, by ensuring that no msg_i matches $shaVal$. To evaluate performance, we make $\psi_{\text{SHA-1}}$ harder to solve by increasing the number of rounds of SHA-1 circuit encoded in the SHA-1$_{circ}$ clauses. We used `SAT-encoding`[3] to generate the SHA-1$_{circ}$ with the different number of rounds (`SAT-encoding` supports 16 to 40 rounds).

We use the same guidance for both SMS and SPECSMS: $\mathsf{NextSplit}() \triangleq (\top, \ell)$. This means that once the secondary solver is active, both SPECSMS and SMS branch on the ℓ variables first. However, SMS does not use speculation. Thus, it only switches to the secondary solver after finding a satisfying assignment to the main solver. In contrast, in SPECSMS, the secondary solver becomes active immediately due to speculation, and, accordingly, the search starts by branching on the ℓ variables.

Results for each set of the queries are shown in Tab. I. Column "# rounds" shows the number of SHA-1 rounds encoded in $\psi_{\text{SHA-1}}$. The problems quickly become too hard for SMS. At the same time, SPECSMS solves all the queries quickly. Furthermore, the run-time of SPECSMS appears to grow linearly with the number of rounds.

The experiments show that prioritizing decisions on ℓ, which is effective in SPECSMS with speculation, is ineffective in SMS. This is expected since this guidance becomes relevant to SMS only after the main solver guessed a satisfying assignment to $\psi_{\text{SHA-1}}$. This, essentially amounts to SMS noticing the guidance only after inverting the SHA-1 circuit, which defeats the purpose of the guidance. As far as we can see, no other guidance can help SMS since there are no good variables to branch on in the main solver and SMS does not switch to the secondary solver until the main solver is satisfied.

VII. Conclusion and Future Work

Modular SAT-solving is crucial for efficient SAT-based unbounded Model Checking. Existing techniques, embedded in IC3/PDR [3] and extended in SMS [1], trade the efficiency of the solver for the simplicity of conflict resolution. In this paper, we propose a new modular SAT-solver, called SPECSMS, that extends SMS with truly bi-directional reasoning. We show that it is provably more efficient than SMS (and, therefore, IC3/PDR). We implement SPECSMS in Z3 [5], extend it with DRUP-style [12] proofs, and proof-based interpolation. We believe this work is an avenue to future efficient SAT- and SMT-based Model Checking algorithms.

In this paper, we rely on user callbacks to guide SPECSMS when to start speculation and (optionally) what variables to prefer in the decision heuristic. This is sufficient to show the power of bi-directional reasoning over uni-directional one of IC3/PDR and SMS. However, this does limit the general applicability of SPECSMS. In the future, we plan to explore guiding speculation by the number of conflicts in the main solver, possibly using similar strategy used for guiding restarts in a modern CDCL SAT-solver [6].

A much earlier version of speculation, called *weak abstraction*, has been implemented in the SPACER Constrained Horn Clause (CHC) solver [10]. Since SPACER extends IC3/PDR to SMT, the choice of speculation is based on theory reasoning. Speculation starts when the main solver is satisfied modulo some theories (e.g., Linear Real Arithmetic or Weak Theory of Arrays). Speculation often prevents SPACER from being stuck in any one SMT query. However, SPACER has no inter-modular propagation and no *refinement*. If *validation* fails, speculation is simply disabled and the query is tried again without it. We hope that extending SPECSMS to theories can be used to make SPACER heuristics much more flexible and effective.

DPLL(T)-style [7] SMT-solvers can be seen as modular SAT-solvers where the main module is a SAT solver and the secondary solver is a theory solver (often EUF-solver that is connected to other theory solvers such as a LIA solver). This observation credited as an intuition for SMS [1]. In modern SMT-solvers, all decisions are made by the SAT-solver. For example, if a LIA solver wants to split on a bound of a variable x, it first adds a clause $(x \leq (b-1) \vee x \geq b)$, where b is the desired bound, to the SAT-solver and then lets the SAT-solver branch on the clause. SPECSMS extends this interaction by allowing the secondary solver (i.e., the theory solver) to branch without going back to the main solver. Control is returned to the main solver only if such decisions tangle local decisions of the two solvers. We hope that the core ideas of SPECSMS can be lifted to SMT and allow more flexibility in the interaction between the DPLL-core and theory solvers.

[2] https://github.com/hgvk94/z3/tree/psms.

[3] Available at https://github.com/saeednj/SAT-encoding.

References

[1] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu. Efficient modular SAT solving for IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 149–156. IEEE, 2013.

[2] N. S. Bjørner, C. Eisenhofer, and L. Kovács. Satisfiability modulo custom theories in Z3. In C. Dragoi, M. Emmi, and J. Wang, editors, *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings*, volume 13881 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2023.

[3] A. R. Bradley. SAT-based model checking without unrolling. In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[4] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[5] L. M. de Moura and N. S. 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.

[6] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[7] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): fast decision procedures. In R. Alur and D. A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[8] A. V. Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008.

[9] E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10886–10891. IEEE Computer Society, 2003.

[10] A. Gurfinkel. Program verification with constrained horn clauses (invited paper). In S. Shoham and Y. Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 19–29. Springer, 2022.

[11] A. Gurfinkel and Y. Vizel. DRUPing for interpolates. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 99–106. IEEE, 2014.

[12] M. Heule, W. A. H. Jr., and N. Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013.

APPENDIX A
SPECSMS AS A SET OF RULES

A configuration of a SPECSMS module is a 4-tuple: $\langle SM, C, \Phi, \mathcal{M}\rangle$, where SM is the search mode (either D^i, P, or F), C is either a (conflict) clause implied by Φ or the marker $none$, Φ is the clause database, and $\mathcal{M}$ is the trail of the solver. The overall configuration of SPECSMS consists of configurations of both of its solvers.

$Initialize$	$\langle P, no, \Phi_{\mathsf{s}}, [\ ]\rangle, \langle D^0, no, \Phi_{\mathsf{m}}, [\ ]\rangle$	
$Prop_{\mathsf{s}}$	$\langle *, no, \Phi_{\mathsf{s}} : (C \vee \ell), \mathcal{M}_{\mathsf{s}}\rangle, NC_{\mathsf{m}} \Rightarrow$ $\langle *, no, \Phi_{\mathsf{s}} : (C \vee \ell), \mathcal{M}_{\mathsf{s}} : \ell^{C \vee \ell}\rangle, NC_{\mathsf{m}}$	ℓ unassigned in $\mathcal{M}_{\mathsf{s}}$, $\neg C \subseteq \mathcal{M}_{\mathsf{s}}$
$Prop_{\mathsf{s}S}$	$\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^X\rangle, \langle *, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow$ $\langle *, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell^X\rangle, \langle *, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell^{C \vee \ell}\rangle$	$\neg C \subseteq \mathcal{M}_{\mathsf{m}}$, ℓ, C shared, $C \vee \ell \in \Phi_{\mathsf{s}}$, ℓ unassigned in $\mathcal{M}_{\mathsf{m}}$
$PropD_{\mathsf{s}S}$	$\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^*\rangle, \langle *, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow$ $\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^*\rangle, \langle *, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell^{\perp}\rangle$	ℓ is shared, ℓ is unassigned in $\mathcal{M}_{\mathsf{m}}$, there does not exists a clause $C \vee \ell \in \Phi_{\mathsf{m}}$ s.t. C is shared and $\neg C \subseteq \mathcal{M}_{\mathsf{s}}$
$Conflict_{\mathsf{s}}$	$\langle *, no, \Phi_{\mathsf{s}} : C, \mathcal{M}_{\mathsf{s}}\rangle, NC_{\mathsf{m}} \Rightarrow$ $\langle *, C, \Phi_{\mathsf{s}} : C, \mathcal{M}_{\mathsf{s}}\rangle, NC_{\mathsf{m}}$	$\neg C \subseteq \mathcal{M}_{\mathsf{s}}$
$Explain_{\mathsf{s}}$	$\langle *, (\neg\ell \vee C), \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^{(\ell \vee D)}\rangle, NC_{\mathsf{m}} \Rightarrow$ $\langle *, (D \vee C), \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^{(\ell \vee D)}\rangle, NC_{\mathsf{m}}$	

TABLE II: Rules independent of solving modes. For each rule, except $Initialize$, there is a symmetrical rule to update Solver m. These rules are not shown here for brevity.$NC_B = \langle *, no, \Phi_{\mathsf{m}}, \mathcal{M}\rangle$

$Decide_{\mathsf{m}}$	$\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle D, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow$ $\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \dagger\rangle, \langle D, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell\rangle$	ℓ unassigned in $\mathcal{M}_{\mathsf{m}}$, ℓ not shared
$Decide_{\mathsf{m}S}$	$\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle D, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow$ $\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \dagger, \ell^{\perp}\rangle, \langle D, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell\rangle$	ℓ unassigned in $\mathcal{M}_{\mathsf{m}}$, ℓ shared
$Reason_{\mathsf{s}}$	$\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^{(\ell \vee C)}\rangle, NC_{\mathsf{m}} \Rightarrow$ $\langle *, (\ell \vee C), \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^{(\ell \vee C)}\rangle, NC_{\mathsf{m}}$	ℓ is shared
$Explain_{\mathsf{ms}}$	$\langle P, C, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle D, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow$ $\langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle D, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle$	C is shared, $C \in \Phi_{\mathsf{m}}$
$Learn_{\mathsf{m}}$	$NC_{\mathsf{s}}, \langle D, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow$ $NC_{\mathsf{s}}, \langle D, C, \Phi_{\mathsf{m}} : C, \mathcal{M}_{\mathsf{m}}\rangle$	$C \notin \Phi_{\mathsf{m}}$
$Learn_{\mathsf{ms}}$	$\langle *, C, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle *, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow$ $\langle *, C, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle *, C, \Phi_{\mathsf{m}} : C, \mathcal{M}_{\mathsf{m}}\rangle$	C is shared $C \notin \Phi_{\mathsf{m}}$
$Backtrack_{\mathsf{m}}$	$\langle *, no, \Phi_{\mathsf{s}}, M_{\mathsf{s}}^k\rangle, \langle D^{i,j}, no, \Phi_{\mathsf{m}}, M_{\mathsf{m}}^k\rangle \Rightarrow$ $\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{k-1}\rangle, \langle D^{i,j}, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{k-1}\rangle$	$k > j$
$Backjump_{\mathsf{m}}$	$\langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle D, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow$ $\langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^i\rangle, \langle D, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^i\rangle$	i is the second highest decision level in $\neg C$
$Fail_{\mathsf{m}}$	$\langle *, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle *, \neg C, \Phi_{\mathsf{m}} : C, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow unsat$	$\neg C$ is decided at level 0

TABLE III: Rules when S_{m} makes decisions and S_{s} does not. † denotes a null literal, a dummy literal that increments the decision level without assigning values to any variable.

FA_{m}	$\langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^i\rangle, \langle D, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^i\rangle \Rightarrow$ $\langle D^i, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^i\rangle, \langle F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^i\rangle$	$\mathcal{M}_{\mathsf{m}}^i$ is a full satisfying assignment to Φ_{m}

TABLE IV: Rule to enter fin mode

$SPECM_{\mathsf{m}}$	$\langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{i}\rangle, \langle D, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{i}\rangle \Rightarrow \langle D^{i}, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{i}\rangle, \langle P, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{i}\rangle$	No clause is unit in Φ_{m} under $\mathcal{M}_{\mathsf{m}}$ and Φ_{s} under $\mathcal{M}_{\mathsf{s}}$

TABLE V: Rule to enter speculation

$Decide_{\mathsf{s}}$	$\langle D, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle D, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \dagger\rangle$	ℓ unassigned in $\mathcal{M}_{\mathsf{s}}$, ℓ not shared
$Decide_{\mathsf{s}S}$	$\langle D, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle P, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle S, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell\rangle, \langle P, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \dagger, \ell^{\perp}\rangle$	ℓ unassigned in $\mathcal{M}_{\mathsf{s}}$, ℓ shared
$Reason_{\mathsf{m}}$	$NC_{\mathsf{s}}, \langle *, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell^{(\ell \vee C)}\rangle \Rightarrow NC_{\mathsf{s}}, \langle *, (\ell \vee C), \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell^{(\ell \vee C)}\rangle$	ℓ is shared
$Explain_{\mathsf{sm}}$	$\langle D, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle P/F, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle D, C, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle$	C is shared, $C \in \Phi_{\mathsf{s}}$
$Learn_{\mathsf{s}}$	$\langle D, C, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, NC_{\mathsf{m}} \Rightarrow \langle D, C, \Phi_{\mathsf{s}} : C, \mathcal{M}_{\mathsf{s}}\rangle, NC_{\mathsf{m}}$	$C \notin \Phi_{\mathsf{s}}$
$Learn_{\mathsf{sm}}$	$\langle D^{i}, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle P, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle D^{i}, no, \Phi_{\mathsf{s}} : C, \mathcal{M}_{\mathsf{s}}\rangle, \langle P, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle$	C is shared $C \notin \Phi_{\mathsf{s}}$
$Backtrack_{\mathsf{s}}$	$\langle D^{i}, C, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{k}\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{k}\rangle \Rightarrow \langle D^{i}, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{k-1}\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{k-1}\rangle$	$k > i$
$Backjump_{\mathsf{s}}$	$\langle D^{i}, C, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle D^{i}, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{k}\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{k}\rangle$	j is the second highest decision level in $\neg C$, $k = max(j, i)$

TABLE VI: Rules when S_{s} is making decisions and S_{m} is not. All rules are symmetric to the ones in Tab. III. The only difference is that there is no counterpart for $Fail_{\mathsf{m}}$. This rule is presented in Tab. VII

$Explain\bot_{\mathsf{s}}$	$\langle D^{i}, (\neg\ell \vee C), \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^{\perp}\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell^{*}\rangle \Rightarrow \langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{i}\rangle, \langle D^{0}, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{i} : \neg\ell\rangle$	ℓ is at a higher decision level than all other literals in C
$Explain\bot_{\mathsf{sm}}$	$\langle D^{i}, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^{*}\rangle, \langle P, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{i}\rangle, \langle D^{0}, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{i} : \neg\ell\rangle$	$\exists m \in C$ s.t $\neg m$ is a local decision in S_{m}, ℓ is shared, ℓ unassigned in $\mathcal{M}_{\mathsf{m}}^{i}$
FA_{A}	$\langle D^{i}, no, \Phi_{\mathsf{s}}, M_{\mathsf{s}}^{j}\rangle, \langle P/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{j}\rangle \Rightarrow \langle F, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{j}\rangle, \langle D^{i,j}/F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{j}\rangle$	$\mathcal{M}_{\mathsf{s}}$ is a full satisfying assignment to Φ_{s}
$Fail_{\mathsf{s}}$	$\langle D^{i}, \neg C, \Phi_{\mathsf{s}} : C, \mathcal{M}_{\mathsf{s}}\rangle, \langle P, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle D^{i}, \neg C, \Phi_{\mathsf{s}} : C, \mathcal{M}_{\mathsf{s}}\rangle, \langle P, \top, \Phi_{\mathsf{m}} : \bot, \mathcal{M}_{\mathsf{m}}\rangle$	$\neg C$ is decided at level 0

TABLE VII: All rules to exit speculation.

$BackjumpV_{\mathsf{m}}$	$\langle F, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle D^{i,j}, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle F, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{k}\rangle, \langle D^{i,j}, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{k}\rangle$	d is the second highest decision level in $\neg C$, $d \geq i$, $k = max(j, d)$

TABLE VIII: Adaptation to the backjump rule for validate mode

$Explain\bot_{\mathsf{m}}$	$\langle F, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}} : \ell^{*}\rangle, \langle D^{i,j}, (\neg\ell \vee C), \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}} : \ell^{\bot}\rangle \Rightarrow \langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{i}\rangle, \langle D^{0}, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{i} : \neg\ell\rangle$	ℓ is at a higher decision level than all literals in C
$BackjumpVE_{\mathsf{m}}$	$\langle F, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle D^{i,j}, C, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle P, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}^{k}\rangle, \langle D^{0}, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}^{k}\rangle$	k is the second highest decision level in $\neg C$, $k < i$
FAV_{m}	$\langle F, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle D^{i,j}, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle \Rightarrow \langle F, no, \Phi_{\mathsf{s}}, \mathcal{M}_{\mathsf{s}}\rangle, \langle F, no, \Phi_{\mathsf{m}}, \mathcal{M}_{\mathsf{m}}\rangle$	$\mathcal{M}_{\mathsf{m}}$ is a full satisfying assignment to Φ_{m}

TABLE IX: All rules to transition out of validate mode