Backtrackable Inprocessing

Alexander Nadel
Faculty of Data and Decision Sciences, Technion, Haifa, Israel

Abstract

We introduce *Backtrackable Inprocessing* (BI), a framework that enables applying inprocessing under the current trail at any decision level, at any point during incremental SAT solving. Our approach lifts the long-standing restriction that inprocessing must be performed only at the global decision level, thereby substantially increasing its potential effectiveness. We focus on three highly efficient core techniques: subsumption, self-subsuming resolution, and Bounded Variable Elimination (BVE). We show how to ensure sound backtracking in the presence of inprocessing, and demonstrate that applying BI for incremental preprocessing after propagating assumptions yields significant performance improvements on Bounded Model Checking (BMC) benchmarks from the Hardware Model Checking Competition 2017. Implemented in the Island SAT solver (IntelSAT's fork), BI enables solving ∼1.5× as many difficult bounds as the baseline global-level incremental preprocessor.

2012 ACM Subject Classification Mathematics of computing → Solvers

Keywords and phrases SAT, Inprocessing, BVE, Bounded Variable Elimination, Subsumption, Incremental SAT

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.27

Related Version *Full Version*: `https://arxiv.org/abs/2605.03654` [14]

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.20034734` [13]

1 Introduction

Modern backtrack-search-based SAT solvers are widely used across numerous applications [5]. A SAT solver determines whether a Boolean formula in Conjunctive Normal Form (CNF) is satisfiable. Since the introduction of MiniSat [9], *incremental* SAT solving has become essential in many crucial domains [1, 2, 7]. In incremental solving, the user performs a sequence of solver queries, each under a set of *assumption literals (assumptions)* [9, 15]. Clauses may be added before any query, and each query checks satisfiability of all clauses provided so far under its specific assumptions. Implementations typically realize this by assigning all assumptions as initial decisions (in order) and propagating after each assignment.

Preprocessing [8] and, later, *inprocessing* [11] have long been known to significantly accelerate SAT solving. Preprocessing simplifies the formula before search, while inprocessing simplifies it regularly during search. One example of an inprocessing technique is *subsumption*: if α and β are clauses with $\alpha \subseteq \beta$, then α subsumes β and β can be removed. Initially restricted to non-incremental solving, inprocessing was adapted for incremental use [16, 10] and has been supported in the mainstream incremental SAT solver CaDiCaL [3] since [10].

However, currently, inprocessing has a crucial limitation: it is applied only *globally at decision level 0*. This restriction significantly limits the power of inprocessing. To illustrate the limitation, consider the clauses $\alpha = a \vee b \vee c$ and $\beta = d \vee b \vee c$. At level 0, α does not subsume β. But if we inprocess at level 1, after assigning $a := 0$, α *does* subsume β – *until the solver backtracks below level 1*. Removing β at this point can be highly beneficial, as it may trigger a chain of further simplifications. This motivates a natural question:

Can inprocessing be invoked under the current trail at any decision level?

No solver offers this capability, with correctness under backtracking the key challenge.

© Alexander Nadel;
licensed under Creative Commons License CC-BY 4.0

1.1 Our Contribution: Backtrackable Inprocessing (BI)

We introduce *Backtrackable Inprocessing (BI)*, the first framework that enables inprocessing under the current trail at any level, at any point in the search. In this work, we focus on three core, extremely effective inprocessing techniques originally applied in the pioneering SatELite preprocessor [8]: subsumption, self-subsuming resolution (which we dub selfsumption), and Bounded Variable Elimination (BVE). BI fully supports incremental SAT solving.

The key challenge is to ensure that the effects of inprocessing are tracked per decision level and can be soundly undone for any level we backtrack over, while remaining in effect at lower levels. We achieve this by generalizing how the solver classifies and manages clauses.

SAT solvers classify clauses into two types: *pervasive* and *temporary* [18, 15]. The set (conjunction) of pervasive clauses is equivalent (or, in some solvers, equisatisfiable) to the set of input clauses, whereas temporaries are implied by the pervasives. Pervasive clauses must be preserved for correctness, while temporary clauses may be deleted at any time (at least at level 0) without affecting correctness and are therefore managed heuristically. Pervasive clauses begin as the input clauses but may change during solving; for example, if a temporary clause α subsumes a pervasive clause β, then β is removed and α becomes pervasive.

We introduce additional clause types for BI, including:

- *stashed at level* i, which are deactivated until the solver backtracks below i, at which point they are restored;
- *deputy at level* i, which remain active and cannot be deleted until backtracking below i.

As an example of how these clause types are created and managed, suppose $\alpha = a \vee b \vee c$ is temporary and $\beta = d \vee b \vee c$ is pervasive under decisions $e := 1$ (level 1) and $a := 0$ (level 2). After detecting that α subsumes β as long as the solver does not backtrack below 2, we mark:

β as a stashed-at-2 clause (deactivated), α as a deputy-at-2 clause (active).

When the solver backtracks below level 2, α will either be deleted or marked as temporary, and β will be restored as pervasive.

In our framework, at all times, each deactivated stashed clause β has a deputy clause α that remains active and *represents* β (by implying it under the current trail) until either β is restored or another stashed clause γ that can represent β is restored. The correct management of stashed and deputy clauses is the core technical contribution of our work.

To further clarify the role of deputy clauses, consider our example above but with α pervasive. Clause β would be handled as before (stashed-at-2 and reactivated as pervasive once the solver backtracks below level 2). The core difference is in the handling of α. In the original example, temporary α must be marked as a deputy to ensure correctness, by guaranteeing that α is not deleted and continues to represent β until β is restored. If α is pervasive, it simply remains so, since pervasive clauses are guaranteed not to be deleted.

We implemented BI in the open-source Island SAT solver, a fork of IntelSAT [12]. While, in principle, BI supports any number of inprocessing steps at arbitrary levels, the empirical part of this work focuses on *incremental preprocessing*. We exploit the fact that BI can inprocess at the beginning of each incremental query *after* assumptions are set and propagated, rather than only at level 0 as in standard incremental inprocessing (e.g., in CaDiCaL [3, 10]). We implemented several BI-enabled incremental preprocessing strategies (Sect. 4.2.4).

We focus on this setting because incremental preprocessing is the core performance-critical component in incremental SAT-based Bounded Model Checking (BMC). BMC plays a central role in hardware validation, where the goal is to verify the design for as many bounds as possible, and solving each new bound may mean the difference between discovering a crucial

bug or missing it entirely. However, each unrolling to a new bound typically results in a substantial increase in the complexity of the SAT instance, making progress toward higher bounds increasingly difficult. Our experiments show that, on BMC benchmarks from the Hardware Model Checking Competition 2017 [6], with a one-hour timeout per benchmark and a bound threshold of 100, Island with standard global-level inprocessing solves 640 more bounds than vanilla Island without inprocessing, whereas BI enables solving 948 more bounds, i.e., about 1.5× more of the difficult bounds unsolved without inprocessing.

Below, Sect. 2–4 give preliminaries, introduce non-incremental subsumption-only BI, and present full BI (proofs in the full version [14]); Sect. 5 and 6 report experiments and conclude.

2 Preliminaries

We begin by recalling standard definitions from propositional logic.

▶ **Definition 1** (Variables, literals, clauses, CNF). *Let $\mathcal{V}$ be a countable set of Boolean variables. A* literal *is either a variable $x \in \mathcal{V}$ or its negation $\neg x$. A* clause *is a finite set (disjunction) of pairwise distinct literals. A CNF formula is a finite set (conjunction) of clauses.*

As is standard in the SAT literature, we represent the SAT solver's search state by a trail.

▶ **Definition 2** (Trail and decision levels). *Let I be an input CNF formula. A* trail *is a finite sequence of literals annotated with their* decision level $\tau = \langle \ell_1@d_1, \ldots, \ell_k@d_k \rangle$, *such that:*

- *all variables occurring in τ are pairwise distinct;*
- *the decision levels satisfy $d_1 \leq \cdots \leq d_k$;*
- *for each i with $d_i > 0$, we say that ℓ_i is a* decision literal *if there is no $j < i$ with $d_j = d_i$; otherwise ℓ_i is an* implied literal*;*
- *writing $\mathrm{Dec}(\tau)$ and $\mathrm{Impl}(\tau)$ for the sets of decision and implied literals on τ, and for $i \in \{1, \ldots, k\}$, $\mathrm{Dec}^{<i}(\tau) := \{ \ell_j \in \mathrm{Dec}(\tau) \mid j < i \}$, we require that for every i with $\ell_i \in \mathrm{Impl}(\tau)$, $I \wedge \bigwedge \mathrm{Dec}^{<i}(\tau) \models \ell_i$.*

In other words, in a valid trail every implied literal $\ell_i \in \mathrm{Impl}(\tau)$ is implied by I together with the decision literals appearing before ℓ_i.

For the remainder of this paper, we fix the input CNF formula to be I and the trail to be τ; both are assumed to be given whenever referenced.

As a direct special case of the trail definition at decision level 0, we get that all global level-0 assignments are implied by the input formula; see Cor. 3.

▶ **Corollary 3** (Global literals implied by I). *Let $\tau = \langle \ell_1@d_1, \ldots, \ell_k@d_k \rangle$ be a trail. For every $i \in \{1, \ldots, k\}$, if $d_i = 0$, then $I \models \ell_i$.*

A trail can be naturally interpreted as a partial assignment consisting of all its literals. For the example trail $\tau = \langle \neg a@0,\ \neg b@1,\ \neg c@2,\ d@2 \rangle$, we have $\tau \models (a \vee b \vee d)$ and $\tau \not\models (a \vee b \vee \neg d)$.

We next extend our terminology to define satisfaction at a given decision level.

▶ **Definition 4** (Restriction of a trail to level d). *For a trail $\tau = \langle \ell_1@d_1, \ldots, \ell_k@d_k \rangle$ and $d \in \mathbb{N}$, define $\tau^{\leq d} := \langle \ell_i@d_i \mid 1 \leq i \leq k,\ d_i \leq d \rangle$, that is, the longest prefix of τ whose decision levels are at most d.*

▶ **Definition 5** (Satisfaction at decision level d). *Let φ be any Boolean formula and $d \in \mathbb{N}$. We say that φ is* satisfied at level d *if $\tau^{\leq d} \models \varphi$.*

We next introduce convenient notation for implication and equivalence evaluated at a given decision level (relative to the fixed input formula I and trail τ).

▶ **Definition 6** (d-implication and d-equivalence). *Let φ, ψ be Boolean formulas, and $d \in \mathbb{N}$. We say that φ* d-implies *ψ, written $\varphi \to^d \psi$, if $\tau^{\leq d} \models (\varphi \to \psi)$, and that φ and ψ are* d-equivalent*, written $\varphi \leftrightarrow^d \psi$, if $\tau^{\leq d} \models (\varphi \leftrightarrow \psi)$.*

For example, given the trail $\tau = \langle e@1,\ \neg a@2\rangle$, we have $(a \vee b \vee c) \to^2 (d \vee b \vee c)$, since $\tau^{\leq 2}$ falsifies a and therefore satisfies $(a \vee b \vee c) \to (d \vee b \vee c)$. In contrast, $(a \vee b \vee c) \to^1 (d \vee b \vee c)$ does **not** hold, as in $\tau^{\leq 1}$ only e is assigned and the implication fails.

Cor. 7 (level-d entailment persists under trail extension) is immediate from Def. 6.

▶ **Corollary 7** (Monotonicity of level-d entailment). *Let φ, ψ be Boolean formulas, and $i, j \in \mathbb{N}$ with $j \geq i$. For $\star \in \{\to, \leftrightarrow\}$, if $\varphi \star^i \psi$, then $\varphi \star^j \psi$.*

For example, with $\tau = \langle e@1,\ \neg a@2,\ g@3,\ \neg h@4\rangle$, we have $(a \vee b \vee c) \to^2 (d \vee b \vee c)$ and hence, by Cor. 7, also $(a \vee b \vee c) \to^3 (d \vee b \vee c)$ and $(a \vee b \vee c) \to^4 (d \vee b \vee c)$.

We now introduce subsumption and its level-dependent variants (relative to the trail τ) and relate them to d-implication, starting with standard subsumption.

▶ **Definition 8** (Subsumption). *Clause α* subsumes *clause β, written $\alpha \subseteq \beta$, if every literal in α also occurs in β.*

▶ **Definition 9** (Pruning falsified literals). *For a clause α and $d \in \mathbb{N}$, we define the* restriction of α to level d *by $\alpha^{\leq d} := \{\ell \in \alpha \mid \neg\ell \notin \tau^{\leq d}\}$, that is, we remove from α all literals whose negation appears on the trail up to level d.*

For the example trail $\tau = \langle e@1, \neg b@2\rangle$, we have $(a \vee b \vee c)^{\leq 2} = (a \vee c)$, whereas $(a \vee b \vee c)^{\leq 1} = (a \vee b \vee c)$.

▶ **Definition 10** (d-subsumption). *Let α, β be clauses and $d \in \mathbb{N}$. We say that α* subsumes *β at d (d-subsumes β), written $\alpha \subseteq^d \beta$, if $\alpha^{\leq d} \subseteq \beta^{\leq d}$.*

For example, under $\tau = \langle \neg e@1,\ \neg b@2\rangle$, we obtain $(a \vee b \vee c) \subseteq^2 (a \vee f \vee c)$, but $(a \vee b \vee c) \not\subseteq^1 (a \vee f \vee c)$ (since b is falsified at level 2 but not at level 1).

It is easy to verify that, as expected, d-subsumption entails d-implication.

▶ **Corollary 11** (d-subsumption entails d-implication). *For any clauses α, β and any $d \in \mathbb{N}$, if $\alpha \subseteq^d \beta$, then $\alpha \to^d \beta$.*

We will need the earliest level d at which a subsumption relation holds, so that a subsumed clause can be safely replaced by the subsuming clause until backtracking below d.

▶ **Definition 12** (min-d-subsumption). *Let α, β be clauses and $d \in \mathbb{N}$. We say that α* minimally subsumes *β at level d (α* min-d-subsumes *β), written $\alpha \subseteq^{\min(d)} \beta$, if:*

1. *$\alpha \subseteq^d \beta$ holds, and*
2. *for every decision level d' strictly smaller than d, $\alpha \subseteq^{d'} \beta$ does not hold.*

Thus d is the minimal decision level at which α d-subsumes β.

For example, with $\tau = \langle e@1, \neg a@2, g@3, \neg h@4\rangle$, we have $(a \vee b \vee c) \to^2 (d \vee b \vee c)$ and hence, by Cor. 7, also $(a \vee b \vee c) \to^3 (d \vee b \vee c)$ and $(a \vee b \vee c) \to^4 (d \vee b \vee c)$. However, $(a \vee b \vee c) \not\to^1 (d \vee b \vee c)$, so the implication at level 2 is minimal, i.e., $(a \vee b \vee c) \subseteq^{\min(2)} (d \vee b \vee c)$.

3 Non-Incremental Backtrackable Inprocessing with Subsumption

This section introduces our framework for backtrackable inprocessing and its main algorithms in the following simplified setting: *non-incremental SAT solving with subsumption as the only inprocessing technique*. This restricted setting already captures the main ideas, terminology, invariants, and algorithms. Section 4 then lifts these restrictions.

We design our Backtrackable Inprocessing (BI) module to be integrated as a component of a SAT solver with a clear API to the main search engine. The search engine interacts with BI via the following two API functions:

1. `Inprocess`: the main inprocessing function. It can be invoked after propagation has completed following either a decision or a conflict analysis, at an arbitrary decision level, and may be called any number of times during the SAT solver invocation.
2. `Backtrack`: let m be the level at the end of the latest `Inprocess` or `Backtrack` invocation. `Backtrack` *must* be invoked once if the solver backtracks to any level below m.

3.1 Clause Classification and Management

Our approach relies on the extended clause classification and management shown in Table 1.

Table 1 Clause classification before the first invocation of `Inprocess` (Table 1a) and after the last `Inprocess` or `Backtrack` invocation ending at level n (Table 1b), as well as additional notation for clause groups (Table 1c). The DL column shows each group's decision level. All clause groups D_i and S_j^q are mutually exclusive. First occurrences of symbols are shown on a light gray background.

(a) Before invoking `Inprocess` for the first time (at any decision level).

DL	Active Clauses	Comment
0	D_0 : active-pervasive	D_0 and I are 0-equivalent ($D_0 \leftrightarrow^0 I$).
∞	D_∞ : temporary	Every clause in D_∞ is implied by D_0 at level 0 ($D_0 \rightarrow^0 D_\infty$).

(b) After `Inprocess` or `Backtrack` invocation ending at level n.

DL	Active Clauses	Stashed Clauses (Inactive)
0	D_0: active-pervasive	
1	D_1 : 1-deputy	$S_1 = S_1^0$
2	D_2 : 2-deputy	$S_2 = S_2^0 \cup S_2^1$
$\vdots$	$\vdots$	$\vdots$
n	D_n : n-deputy	$S_n = S_n^0 \cup S_n^1 \cup \cdots \cup S_n^{n-1}$
∞	D_∞: temporary	

(c) Additional notation for clause groups. Note that $S^i \subseteq S$ for every i, and $D \subseteq N \subseteq A$.

Symbol	Name	Definition	Active?
S	stashed	$S_1 \cup S_2 \cup \cdots \cup S_n$	No
S^i	i-restored	$S_1^i \cup S_2^i \cup \cdots \cup S_n^i$	No
P	pervasive	$D_0 \cup S^0$	Mixed
D	deputy	$D_1 \cup \cdots \cup D_n$	Yes
N	active-non-temporary	$D_0 \cup D$	Yes
A	active	$N \cup D_\infty$	Yes
I	input	the original input clauses	

BI invariants (see Table 1 for notation)

1. **Input-equivalence:** $\boldsymbol{I \leftrightarrow^0 P}$**.** *In words:* the current set of pervasive clauses is logically equivalent to the original input clauses.
2. **Correctness:** $\boldsymbol{N \leftrightarrow^n P}$**.** *In words:* the active-non-temporary clauses are logically equivalent to the pervasive clauses until backtracking below n.
3. **Representation.** For all $\alpha \in S_{i \le n}$, one of the following holds:
   a. **Deputy-representation**: For some $r \le i$ and r-deputy $\beta \in D_r$, $\beta \rightarrow^i \alpha$.
   b. **Restored-representation**: For some $r \le i$, $j > i$, and r-restored $\beta \in S^r_j$, $\beta \rightarrow^i \alpha$.
   c. **Trivial-representation**: $\top \rightarrow^i \alpha$ (equivalently, $\tau^{\le i} \models \alpha$ by Def. 5).
   In words: every clause stashed at some level $i \le n$ is implied at level i either by a deputy whose level is at most i, or by a clause stashed at some level $j > i$ that will be restored as such a deputy, or else is trivially implied at level i.
4. **Closure:** $\boldsymbol{\forall \gamma \in (A \cup S).\, P \rightarrow^0 \gamma}$**.** *In words:* every clause, active or stashed, is implied by the pervasive clauses.

Figure 1 BI invariants (see Table 1c for notation). The level n is the decision level of the latest completed BI invocation (`Inprocess` or `Backtrack`). The connectives $\rightarrow^i$ and $\leftrightarrow^i$ denote implication and equivalence at decision level i (Def. 6). Inv. 2 follows from the other invariants (Lemma 13).

Table 1a demonstrates that, before the initial invocation of `Inprocess` (at whatever decision level), the solver maintains two standard clause types: the active-pervasive clauses D_0, which are logically equivalent to the original input clauses I up to level-0 simplifications, and the temporary (learnt) clauses D_∞, each of which is implied by D_0 at level 0.

Table 1b and Table 1c present the clause classification from the first invocation of `Inprocess` onward, generalizing and extending this standard partition.

Specifically, in addition to active-pervasive and temporary clauses, we track inactive *stashed* clauses and active *deputy* clauses. Before restoration, each stashed clause is always represented by some active deputy clause, although this deputy may change over time.

Each active clause group $D_k \subseteq A$ and each stashed clause group $S^q_k \subseteq S$ is indexed by a level k. Clauses in D_k remain active and *must not be removed* until the solver backtracks below k. Once the solver backtracks below k, this restriction is lifted, and clauses in D_k may either be deleted or treated as temporary clauses, that is, moved to D_∞. Since the solver is always below ∞, clauses in D_∞ are, by convention, always eligible for deletion.

Clauses in a stashed group S^q_k remain inactive until the solver backtracks below their associated level k. When this occurs, they are restored into D_q, where we refer to $q < k$ as their *restoration level*.

3.2 Invariants

Fig. 1 summarizes the invariants jointly maintained by the main SAT search engine and our BI module. The BI module is explicitly designed to enforce these invariants, while they serve as requirements for the search engine. We will shortly argue that these requirements are natural for a standard SAT solver.

3.2.1 High-Level Invariants and Framework Correctness

Invs. 1 and 2 together guarantee the correctness of our framework: at any point during the search, the active-non-temporary clauses N are logically equivalent to the original input clauses I (up to level-0 simplifications implied by I). Strictly speaking, Inv. 2 ensures this equivalence only until backtracking below level n, but maintaining this invariant during backtracking is precisely the responsibility of our `Backtrack` procedure.

3.2.2 Low-Level Invariants

If Invs. 1 and 2 capture the high-level correctness of our framework, then Invs. 3 and 4 provide the corresponding low-level guarantees.

Intuitively, Inv. 3 ensures that every stashed clause is represented until its restoration. Unless it is trivially implied (trivial-representation), it is either represented by an active deputy (deputy-representation) or by another stashed clause that will later be restored as such a deputy (restored-representation). In the restored-representation case, at any concrete point the stashed clause is still represented by an active deputy, but *transitively*: it is represented via a chain of stashed clauses, the last of which is represented by an active deputy.

Inv. 4 ensures that all clauses in the system are implied by the pervasive clauses.

Lemma 13 shows that Invs. 3 and 4 imply Inv. 2 throughout the execution of our algorithm. The proofs of this and all other lemmas appear in the full version [14].

▶ **Lemma 13** (Correctness from representation and closure). *Assume Invs. 3 and 4 hold. Then Inv. 2 holds as well, i.e.,* $N \leftrightarrow^n P$.

3.2.3 Why the Search Engine Maintains All Invariants

We argue that a standard SAT search engine naturally maintains our invariants, both prior to the first `Inprocess` call and between subsequent invocations of `Inprocess` and `Backtrack`. Sect. 3.3 constructively shows that our BI module maintains them as well.

We assume that the search engine itself does not apply inprocessing (which is instead handled via interaction with our BI module), and that it performs only standard level-0 simplifications implied by the input clauses I, such as deleting globally satisfied clauses and removing globally falsified literals. Throughout this section, all logical equivalences and implications are understood modulo these level-0 simplifications, as justified by Cor. 3.

Let us verify that our invariants (skipping Inv. 2 by Lemma 13) hold at the time of the first `Inprocess` call by the search engine. Inv. 1 holds since, by construction, D_0 comprises the input clauses I up to the standard level-0 simplifications. Inv. 3 is vacuous, because no stashed clauses exist before inprocessing. Inv. 4 holds since P comprises D_0, and $A \cup S$ comprises $D_0 \cup D_\infty$, where every temporary clause in D_∞ is implied by D_0 by construction.

We now argue that the search engine maintains all invariants in each phase between two BI invocations (`Inprocess` or `Backtrack`). Let n be the decision level of the latest BI call. Crucially, during the phase, the level never drops below n, since `Backtrack` is invoked whenever backtracking below n occurs. At the beginning of the phase, Invs. 1 and 2 imply that I, P, and N are equivalent at level n, while D_∞ is implied by these groups by Inv. 4. The search engine operates only on the active clauses $A = N \cup D_\infty$, which, by construction, remain equivalent to I at level 0 until the end of the phase. Hence Invariants 1 and 4 continue to hold throughout the phase. As for Inv. 3, the stashed clause sets do not change, being inactive, while deputy clauses may either remain unchanged or be simplified at level 0 (the latter only if the solver allows level-0 simplifications at arbitrary levels). Such simplifications

preserve restored- and trivial-representation. Deputy-representation is preserved, except when a representing deputy becomes satisfied at level 0 and is deleted, in which case any stashed clause it represented falls under trivial-representation. Inv. 2 holds by Lemma 13.

3.3 Algorithms

In this section, we introduce our BI algorithms `Inprocess` and `Backtrack`, and demonstrate that they maintain the invariants in Fig. 1.

3.3.1 `Inprocess`

Algorithm 1 `Inprocess` at level n.

1: **while** there exist distinct active clauses $\alpha, \beta \in A$ and some l such that $\alpha \subseteq^{l} \beta$ **do**
2: Pick any active $\alpha \in D_a$ and $\beta \in D_b$ with $0 \le a, b \le \infty$, $\beta \neq \alpha$, such that $\alpha \subseteq^{\min(k)} \beta$
3: **if** $b < k$ **then**
4: **if** $a > k$ **then** Promote α to D_k
5: Deactivate and stash β by moving it to S_k^b
6: **else** ▷ $b \ge k$
7: **if** $a > b$ **then** Promote α to D_b
8: Delete β ▷ Alternatively, demote β to a temporary clause in D_∞

Our `Inprocess` algorithm is shown in Alg. 1. It operates in a loop. In each iteration, it selects a pair of distinct active clauses $\alpha \in D_a$ and $\beta \in D_b$ with $0 \le a, b \le \infty$ such that α subsumes β at some decision level, and this subsumption is *minimal* at level k. (If no such pair exists, the algorithm terminates.) The goal is either to delete β entirely whenever this is sound, or otherwise to stash β for as long as possible – that is, with the lowest possible restoration level – while ensuring that β remains represented until it is restored.

Before presenting the algorithm in detail, we need a notion of rank; see Def. 14.

▶ **Definition 14** (Rank of an active clause). *The* rank *of an active clause $\gamma \in D_i$ is i.*

Promoting γ to a lower group (i.e., moving γ from D_i to some D_j with $j < i$) does not break the invariants in Fig. 1, and in particular preserves Inv. 3, as formalized in Lemma 15.

▶ **Lemma 15** (Promotion preserves invariants). *Let $\gamma \in D_i$ be an active clause, and suppose all invariants in Fig. 1 hold. If we* promote *γ to some group D_j with $j < i$ (by moving it), then all invariants continue to hold.*

We now analyze one iteration of `Inprocess` by case-splitting on the relationship between β's rank b and the minimal subsumption level k.

If β's rank b is lower than k (line 3), we deactivate and stash β at level k with b as its restoration level (line 5). Before that, if α's rank is higher than k, we promote α to D_k (line 4) so that it can represent β while the solver remains at levels $\ge k$. Furthermore, not only β itself then becomes represented by α, but so does any clause stashed at level k or higher that was previously represented by β, whereas any clause stashed at level $< k$ and previously represented by β is still represented by β, but now via restored-representation. Lemma 16 formalizes that these updates preserve all BI invariants.

▶ **Lemma 16** (Soundness of the $b < k$ case). *In the case $b < k$, the updates in lines 4–5 preserve all BI invariants in Fig. 1.*

Otherwise, **if β's rank b is not lower than k**, we promote α to D_b if its rank a is not already $\leq b$ and delete β (or demote β to a temporary clause in D_∞). This is safe because α then represents both β and any clause previously represented by β until backtracking below b. Once the solver backtracks below b, Cor. 17 implies that β no longer represents any clause. Hence, β can be safely deleted. Cor. 17 follows from Def. 14 and Inv. 3.

▶ **Corollary 17** (No representation below rank). *An active clause of rank k can only represent stashed clauses at levels $\geq k$.*

Lemma 18 shows that these updates preserve all BI invariants.

▶ **Lemma 18** (Soundness of the $b \geq k$ case). *In the case $b \geq k$, the updates in lines 7–8 preserve all BI invariants in Fig. 1.*

3.3.2 `Backtrack`

Algorithm 2 `Backtrack` to level n.

```
1: Let n be the current decision level (after backtracking).
2: Let m ← max{ i | D_i ≠ ∅ or S_i ≠ ∅ }        ▷ Highest level with active or stashed clauses
3: for i ← m downto n + 1 do
4:     for all α ∈ D_i do Delete α ▷ Alternatively, demote α to a temporary clause in D_∞
5:     for all q ∈ {0, ..., i − 1} do
6:         for all β ∈ S_i^q do Activate β and move it from S_i^q to D_q
```

Our backtracking algorithm `Backtrack` is shown in Alg. 2. Given the current decision level n after the SAT solver has backtracked, the procedure identifies the highest level m at which there are still active or stashed clauses, and then iterates from m down to $n+1$. For each visited level i, it first deletes all deputy clauses in D_i, which is sound (w.r.t. Inv. 3) because no clause of rank i can represent any stashed clause below its rank (Cor. 17). Alternatively, such deputies may be demoted to temporary clauses, without affecting correctness (Inv. 4). It then restores every clause stashed at level i with restoration level q by moving it into D_q, which is safe w.r.t. Inv. 3 because any stashed clause previously restored-represented via a freshly restored clause continues to be represented by it, now via deputy-representation.

Lemma 19 formally argues that a single iteration of the loop in Alg. 2 (i.e., processing one level i by deleting D_i and restoring all S_i^q) preserves all BI invariants in Fig. 1. By induction over the iterations $m, m-1, \ldots, n+1$, this yields the correctness of `Backtrack`.

▶ **Lemma 19** (One-step backtrack preserves invariants). *Processing a single level i in `Backtrack` (i.e., deleting D_i and restoring all S_i^q into D_q) preserves all BI invariants in Fig. 1.*

4 Backtrackable Inprocessing

This section extends the subsumption-only non-incremental BI algorithm to support self-sumption, BVE, and incremental SAT solving. With the theoretical foundations (clause classification, invariants, correctness) already established in Sect. 3, we focus primarily on the practical aspects of the algorithms.

Our high-level inprocessing algorithm largely reuses *incremental SatELite* – the original global level-0 non-incremental SatELite preprocessing algorithm [8], extended to global level-0 *incremental* preprocessing in our earlier work [16]. Sect. 4.1 reviews incremental SatELite and related work, and Sect. 4.2 presents our complete BI algorithm.

4.1 Incremental Preprocessing with SatELite and Related Work

We start with standard definitions for resolution and selfsumption (self-subsuming resolution).

▶ **Definition 20** (Resolution). *Let α, β be clauses and x a variable. They are* resolvable on x *if $x \in \alpha$ and $\neg x \in \beta$. In this case the* resolvent *is $\alpha \otimes_x \beta := (\alpha \setminus \{x\}) \cup (\beta \setminus \{\neg x\})$, and the* resolution rule *derives $\alpha \otimes_x \beta$ from α and β.*

▶ **Definition 21** (Selfsumption, Strengthening). *Let α, β be clauses resolvable on x. We say that α* selfsumes *β on x if $\alpha \otimes_x \beta \subseteq \beta$. Equivalently, writing $\alpha = C \vee x$ and $\beta = D \vee \neg x$, this holds iff $C \subseteq D$. In this case, β may be* strengthened *to $D = \alpha \otimes_x \beta$.*

For example, if $\alpha = (a \vee x)$ and $\beta = (a \vee b \vee \neg x)$, then the resolvent on x is $\alpha \otimes_x \beta = (a \vee b)$, which is a subset of β. Hence α selfsumes β on x, and β can be strengthened to $(a \vee b)$.

Finally, we use the term *sumption* to refer to subsumption and selfsumption together, and the verb *to sume* to indicate that a clause either subsumes or selfsumes another clause.

Alg. 3 presents incremental preprocessing with SatELite [8, 16] in high-level, using the core data structures listed in Fig. 2.

Consider the data structures in Fig. 2. The occurrence lists $\text{occ}[\ell]$ store all clauses containing literal ℓ. The clause heap $\mathcal{H}$ is a size-based heap used for efficient backward sumption to scan candidate clauses from smallest to largest to see whether they sume others. Finally, all eliminated variables are maintained in their elimination order in $\mathcal{O}$ (preserving this order is crucial for incrementality [16]), and for each eliminated variable v the original clauses are stored in $F^+(v)$ and $F^-(v)$.

Alg. 3 is invoked at the beginning of every incremental SAT query. It first scans the clauses to initialize occ and $\mathcal{H}$ (line 1). Then, the loop in line 2 iterates over all eliminated variables in $\mathcal{O}$. If new clauses containing a variable v have appeared since the previous invocation, the algorithm chooses to either reintroduce v by restoring $F^+(v)$ and $F^-(v)$ (optionally removing their resolvents) and removing v from $\mathcal{O}$, or reeliminate v by adding the new clauses to $F^+(v)$ and $F^-(v)$ and creating new non-tautological resolvents, using the standard bounded-clause elimination heuristic to make this choice. Both reintroduction and reelimination may generate new clauses with eliminated variables, but, crucially for correctness, only with variables that will still be visited later in the current pass over $\mathcal{O}$ [16, 10]. The algorithm then enters the main preprocessing loop (line 5). It begins by applying backward sumption to a fixpoint (line 6). Next, every variable that occurs in clauses removed or updated during the latest sumption step is considered for elimination (line 7). A variable v is eliminated as long as doing so does not increase the clause count (line 8). Finally, elimination is applied only to pervasive clauses: temporary clauses are ignored, and, if elimination succeeds, any temporary clauses containing v are deleted.

4.1.1 Other Related Work

We reuse the high-level structure of incremental SatELite in our BI framework. Supporting gate extraction [8], as well as extending BI beyond sumption and variable elimination (in the spirit of [10], which achieves this for global level-0 inprocessing), is left for future work. We previously proposed incremental preprocessing after propagating assumptions in [17]. However, the present work is both significantly more general – our framework supports *in*processing at any decision level and at any point during solving, rather than only *pre*processing at the beginning of each incremental query – and practically more useful, since [17] is closed-source (whereas our implementation is open-source) and relies on maintaining resolution proofs in memory, an infrastructure not supported by modern SAT solvers.

Core data structures for incremental preprocessing with SatELite [8, 16]

- *Per-literal occurrence lists* occ[ℓ]: vector of clauses containing literal ℓ.
- *Clause heap* $\mathcal{H}$: size-based clause heap for backward sumption.
- *Eliminated variables:*
  - **(i)** eliminated variables in their elimination order $\mathcal{O} = [v_1, \ldots, v_m]$;
  - **(ii)** for each $v \in \mathcal{O}$, original clauses $F^+(v) = \{C \mid v \in C\}$ and $F^-(v) = \{C \mid \neg v \in C\}$.

Figure 2 Core data structures used by incremental SatELite.

Algorithm 3 Incremental Preprocessing with SatELite.

```
1:  Initialization: scan clauses to initialize occ and H.
2:  for each v ∈ O do                          ▷ Reintroduction/reelimination loop
3:      if new clauses with v exist then
4:          Reintroduce or reeliminate v       ▷ Reintroduction removes v from O
5:  repeat                                     ▷ Main preprocessing loop
6:      Apply backward sumption (subsumption + selfsumption) to a fixpoint.
7:      for each variable v in clauses removed or updated during latest sumption do
8:          if eliminating v does not increase the clause count then  ▷ ignoring temporaries
9:              Eliminate v: save F+(v) ∪ F−(v) and add non-tautological resolvents.
10:             Append v to O.
11: until no elimination in the last iteration
```

4.2 Incremental Backtrackable Inprocessing with Sumption and BVE

Throughout the rest of the paper, whenever we refer to level-0 notions (satisfaction, implication, equivalence, or subsumption with respect to $\tau^{\leq 0}$), we call them *global*. In contrast, notions at levels $d > 0$ (i.e., with respect to $\tau^{\leq d}$ for some $d > 0$) are called *conditional*, since they depend on the current trail.

To integrate our subsumption-only BI framework from Section 3 into the SAT solver, it is sufficient to:

1. Apply `Inprocess` (Alg. 1) for handling conditional subsumption directly as the implementation of the sumption step in incremental SatELite (line 6 in Alg. 3), retaining the original processing order from smallest to largest clauses via the size-based heap, but ignoring conditionally falsified literals when computing clause sizes.
2. Stay synchronized with respect to backtracking: let m be the decision level at the end of the latest `Inprocess` or `Backtrack` (Alg. 2) invocation. Then `Backtrack` *must* be invoked whenever the solver backtracks below level m.

Below, Sect. 4.2.1 defines conditional selfsumption and explains how to extend the above BI integration in incremental SatELite to support it, while Sects. 4.2.2 and 4.2.3 show how to handle conditionally satisfied clauses and BVE, respectively, within our framework. Finally, Sect. 4.2.4 discusses the integration of incremental SAT solving (which comes for free in our framework) and introduces several strategies for applying BI for incremental preprocessing.

4.2.1 Handling Selfsumption

We begin by defining conditional d-selfsumption and min-d-selfsumption, in analogy to d-subsumption (Def. 10) and min-d-subsumption (Def. 12).

▶ **Definition 22** (d-selfsumption). *Let α, β be clauses resolvable on x and $d \in \mathbb{N}$. We say that α d-selfsumes β on x, written* $\mathrm{selfsume}_x^d(\alpha, \beta)$*, if $(\alpha \otimes_x \beta)^{\leq d} \subseteq \beta^{\leq d}$.*

▶ **Definition 23** (min-d-selfsumption). *Let α, β be clauses resolvable on x and $d \in \mathbb{N}$. We say that α* minimally *d-selfsumes β on x, written* $\mathrm{selfsume}_x^{\min(d)}(\alpha, \beta)$*, if:*
1. $\mathrm{selfsume}_x^d(\alpha, \beta)$ *holds, and*
2. *for every decision level d' strictly smaller than d,* $\mathrm{selfsume}_x^{d'}(\alpha, \beta)$ *does not hold.*

Thus d is the minimal decision level at which α d-selfsumes β on x.

We have augmented Alg. 1 to support d-selfsumption as follows:
1. We identify min-d-selfsumption alongside min-d-subsumption.
2. If $d = 0$ (selfsumption is global), it is handled by strengthening as in global inprocessing.
3. Whenever conditional $\mathrm{selfsume}_x^{\min(d)}(\alpha, \beta)$ for $d > 0$ is discovered, we reduce its handling to d-subsumption as follows. We create a new clause marked as *temporary*, $\gamma \in D_\infty$, with $\gamma := \alpha \otimes_x \beta$. By construction, $\gamma \subseteq^{\min(d)} \beta$ holds, hence we let our d-subsumption algorithm handle this case.

We emphasize that, unlike in global selfsumption, in conditional selfsumption we cannot soundly strengthen the selfsumed clause by removing a literal; instead, we must create a new clause. This is due to the presence of conditionally falsified literals in the selfsuming α but not in the selfsumed β, rendering the strengthening justified only conditionally and not globally. For example, let $\alpha \in A$, $\alpha = (a \vee b \vee x)$, and $\beta \in N$, $\beta = (b \vee c \vee \neg x)$ under the trail $\tau = \langle \neg a@1 \rangle$. We have $\alpha \otimes_x \beta = (a \vee b \vee c)$, so $(\alpha \otimes_x \beta)^{\leq 1} = (b \vee c)$ and $\beta^{\leq 1} = (b \vee c \vee \neg x)$, hence $(\alpha \otimes_x \beta)^{\leq 1} \subseteq \beta^{\leq 1}$ and thus α min-1-selfsumes β on x. Our d-selfsumption mechanism therefore creates a new temporary clause $\gamma \in D_\infty$ with $\gamma := \alpha \otimes_x \beta = (a \vee b \vee c)$ and, given that $\gamma \subseteq^{\min(1)} \beta$, delegates the rest to d-subsumption, which stashes β at level 1 and promotes γ to D_1 to represent β. Clearly, removing $\neg x$ from β would be unsound.

As a practical note, since creating new clauses is significantly more expensive than removing literals from existing ones, our implementation applies conditional selfsumption only when the number of unassigned literals in the selfsumed clause is at most 3.

4.2.2 Handling Conditionally Satisfied Clauses

Unlike in global inprocessing, backtrackable inprocessing may encounter clauses that are *conditionally* satisfied under the current trail and thus irrelevant at the current decision level. To handle such clauses, we first define minimal satisfaction (w.r.t. the implicit trail τ).

▶ **Definition 24** (Minimal satisfaction). *Let φ be a Boolean formula. We say that φ is* minimally satisfied at level d *under τ if:*
1. *φ is satisfied at level d under τ, i.e., $\tau^{\leq d} \models \varphi$, and*
2. *for every decision level d' with $0 \leq d' < d$, we have $\tau^{\leq d'} \not\models \varphi$.*

In this case, d is the minimal decision level at which φ is satisfied under τ.

We handle conditionally satisfied clauses as follows. During the inprocessor's initialization (line 1 of Alg. 3), we do *not* insert them into the main data structures (the occurrence lists occ and the clause heap $\mathcal{H}$). Instead, taking advantage of our clause classification, whenever a clause $\beta \in D_i$ is conditionally satisfied under the current trail, we compute its minimal

satisfaction level d (in the sense of Def. 24), remove β from the active set D_i, and stash it in S_d^i to be restored once the solver has backtracked below d, by which time β will no longer be satisfied. Note that Inv. 3 is maintained for any such clause due to trivial-representation.

4.2.3 Handling Bounded Variable Elimination (BVE)

We now turn to Bounded Variable Elimination (BVE). Unlike for sumption, there is no "conditional" variable elimination, since the variable elimination algorithm itself does not depend on the trail. However, the presence of conditionally satisfied clauses is promising for *bounded* variable elimination *efficiency-wise*: skipping such clauses as described in Sect. 4.2.2 increases the chances of eliminating more variables while adhering to the clause-count bound. For correctness, when those clauses are later restored, the relevant eliminated variables must be reintroduced or reeliminated, but this is handled automatically as laid out below.

Our BI integration preserves the standard BVE scheme of incremental SatELite, but, as explained below, it introduces six easy-to-implement, BI-specific changes, mostly related to assigning and managing ranks of original clauses and resolvents. Below, these changes are numbered as **(Ci)** at the end of the sentences introducing them.

In BI, new clauses containing eliminated variables may arise not only from the user (as in standard incremental SAT), but also from restoring stashed clauses. Fortunately, incremental SatELite (Alg. 3) already handles clauses with eliminated variables regardless of their origin. We just need to ensure that the search algorithm invokes inprocessing whenever even one clause with an eliminated variable is present in the solver, whether it is a user clause added during a new incremental query or a restored stashed clause **(C1)**.

The remaining main question for handling BVE in BI is how to treat deputy clauses during elimination and backtracking. Recall that a deputy of rank i (that is, a deputy in D_i) is required only as long as the solver does not backtrack below i, at which point all clauses in D_i are deleted by `Backtrack` (Alg. 2). Therefore, conceptually, any resolvents produced using such deputies become obsolete as soon as the solver backtracks below the highest rank of their parents. To reflect this, we assign each resolvent the *maximum* rank of its parent clauses as follows: when, while eliminating a variable v, we resolve $\alpha \in D_i$ with $\beta \in D_j$ on v, producing the resolvent $\gamma = \alpha \otimes_v \beta$, we assign γ to D_k, where $k := \max(i, j)$ **(C2)**. Furthermore, we keep the original rank of every clause in $F^+(v)$ and $F^-(v)$ for any eliminated variable v **(C3)**. When the solver backtracks below level i, all clauses of rank i are deleted from $F^+(v)$ and $F^-(v)$ for every eliminated variable v **(C4)**. When a variable v is reintroduced, the remaining clauses in $F^+(v)$ and $F^-(v)$ keep their rank when activated **(C5)**. Finally, as BI is designed to solve the problem under the trail, BVE attempts to eliminate only unassigned variables, ignoring those already assigned on the trail **(C6)**.

Furthermore, following prior work on incremental inprocessing [16, 10], we restrict elimination to variables occurring in newly added clauses; in BI, we further consider variables appearing in clauses that contain literals assigned on the current trail, thereby avoiding repeated elimination attempts for variables whose relevant context has not changed.

4.2.4 Incrementality and BI Strategies for Incremental Preprocessing

BI inherently accommodates clause additions and changing assumptions, so our framework requires no modifications to support incremental SAT solving and can be applied as is.

Although BI supports any number of inprocessing steps at arbitrary levels, we focus on studying the effects of *incremental preprocessing* after the assumptions are propagated – a minimal step that already distinguishes BI from standard global incremental inprocessing, both in terms of algorithmic capabilities and in its empirical impact on Bounded Model Checking (BMC). Below we present several strategies for incremental preprocessing.

We call the first incremental preprocessing strategy enabled by BI the **Assumption-Level (AL) strategy**. It applies inprocessing immediately after propagating the assumptions for every incremental SAT query. As a baseline for comparison, we also implement a global-level **BASE strategy** that backtracks to 0 and preprocesses there with the assumptions frozen (i.e., not allowed to be eliminated) at the start of every incremental query (as in CaDiCaL [3]).

Initial experiments on BMC instances showed that, although the BI-enabled AL is often more efficient, there are benchmarks where the standard global-level BASE wins. Further analysis revealed that the reason lies in how variables of literals implied by assumptions, and clauses that become conditionally satisfied under assumptions, are treated.

AL ignores assigned variables and stashes conditionally satisfied clauses before inprocessing, which can be beneficial because less data is processed and BVE can eliminate more variables. However, there is a trade-off. First, eliminating variables that are about to be implied by assumptions can be helpful, and this is carried out by BASE but not AL. Second, AL misses selfsumptions by conditionally satisfied clauses that could remove falsified literals from other clauses, as illustrated below. Let $\alpha = (a \vee b \vee c)$ and $\beta = (\neg a \vee b \vee c)$ be clauses handled by AL under the trail $\tau = \langle a@1 \rangle$. Since α is stashed by BI as satisfied, AL misses the opportunity to strengthen β to $(b \vee c)$. While this does not affect reasoning under the current trail, any new temporary clauses that depend on β (including clauses selfsumed by β during inprocessing and learnt clauses during search) must carry $\neg a$, inflating the clause database and causing cache misses. In contrast, BASE would have strengthened β.

Hence, we introduce a combined **COMB strategy**: immediately after a new SAT query, we first run inprocessing at level 0 (BASE) and then again after assigning and propagating assumptions (AL). This layered approach, despite its own overhead, can sometimes simplify the formula more effectively by first applying global and then conditional simplifications.

Finally, we observed a clear pattern in when AL, BASE, and COMB work best. Let M be a measure of how intensively BI is used while AL is running. By default, we define $M = \frac{\text{number of conditional selfsumptions}}{\text{number of inprocessing invocations}}$ (a slightly less effective variant is $\frac{\text{stashed clause restorations}}{\text{number of conflicts}}$). Empirically, on instances with a small M, AL tends to be most effective (it is cheap while still having noticeable impact); for medium values of M, BASE is preferable (the additional overhead of AL is not justified); and when M is large, COMB works best (it is then worthwhile to invest more in inprocessing, given its strong overall effect).

Hence, we designed the following dynamic **DYN strategy**. We start with AL and, after 10 000 conflicts, permanently select a strategy based on the value of M: if $M < 200$, we stay with AL; if $200 \leq M < 3000$, we switch to BASE; and if $M \geq 3000$, we switch to COMB.

5 Experimental Results

We implemented BI in the Island SAT solver, a fork of IntelSAT [12].

We started from the 300 bit-level benchmarks of the Hardware Model Checking Competition 2017 [6]. We unrolled each benchmark to bound 100 using the `aigbmc` tool [4], to obtain incremental SAT instances, and dumped them in IntelSAT's incremental SAT format (an extension of DIMACS) [12]. For two of the benchmarks, unrolling ran out of memory beyond mid-50 bounds, so we excluded these benchmarks and used the remaining 298 in our experiments. To simulate BMC, we instructed the SAT solver to stop after the first satisfiable query, if any.

For all experiments, we used machines with 256 GB of memory and Intel® Xeon® E5-2690 v4 processors running at up to 3.5 GHz, and we set the timeout to one hour. The solver and the benchmarks are publicly available [13].

We compare the three BI-enhanced incremental preprocessing strategies AL, COMB, and DYN against the baseline BASE, which implements standard global incremental preprocessing, in terms of the number of solved bounds. To filter out trivially solved, uninteresting bounds, we report the delta relative to the vanilla Island without inprocessing.

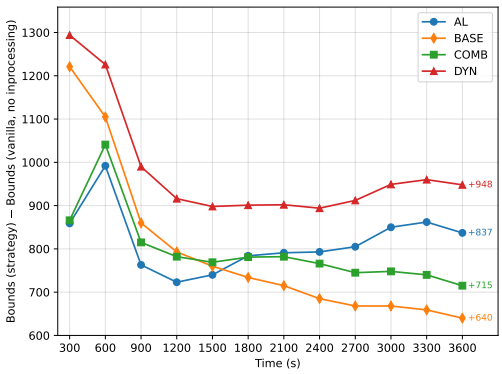


Figure 3 Bounds solved over time per strategy: delta vs. vanilla (no inprocessing).

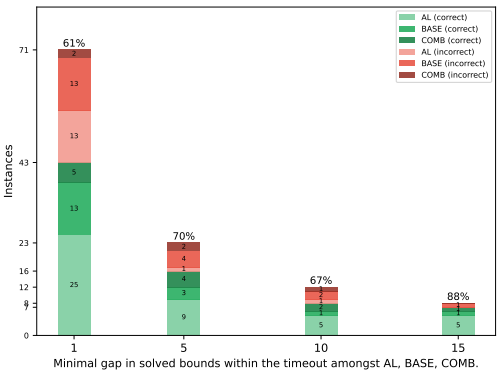
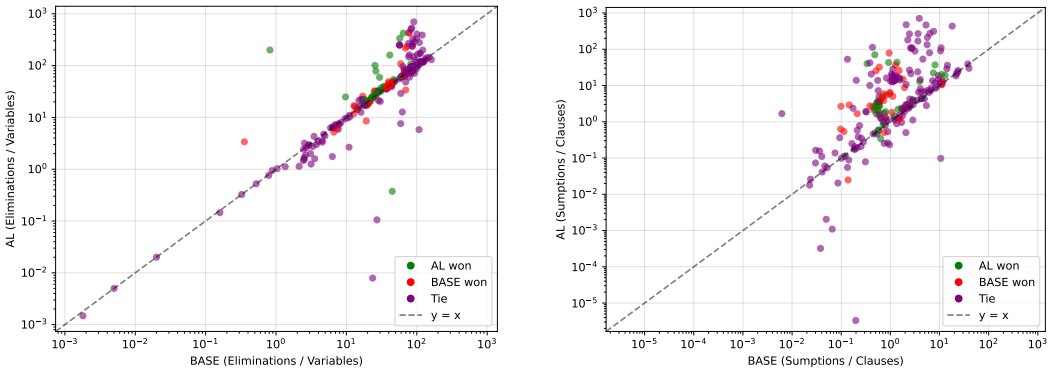


Figure 4 DYN quality per minimal gap in solved bounds amongst AL, BASE, COMB.

The main result is shown in Fig. 3, which reports, over time, the total number of bounds solved per strategy (delta vs. vanilla). Although the simplest BI strategy AL (that performs inprocessing after assumptions are propagated) is initially outperformed by the baseline BASE, AL eventually overtakes BASE and, at the timeout, solves 197 more bounds. DYN is the superior strategy: it is more efficient than BASE from the very beginning with the gap widening to 308 more bounds than BASE and 111 more bounds than AL at the timeout. COMB is not as effective as the two other BI-enabled strategies overall.

Consider Fig. 4, which analyzes the quality of DYN's strategy choices. Each vertical bar at $x = n$ shows the decision quality of DYN on contested instances where AL, BASE, and COMB achieve different numbers of solved bounds; here, n is the gap between the highest and lowest of those counts, so larger n means higher stakes for a correct decision by DYN. The percentage above each bar is the fraction of those instances on which DYN chooses correctly. Within each bar, green segments correspond to a correct choice (a strategy that achieves the maximum number of solved bounds), and red segments to an incorrect choice; sub-segments indicate which strategy was chosen (AL, BASE, or COMB). As the gap increases, DYN's correctness rate rises and reaches 88% for the largest gap. Thus, DYN tends to choose correctly precisely when the payoff – the difference between the best and worst static strategy – is highest. Although COMB is outperformed by the other static strategies overall, it is still sometimes chosen by DYN even for the highest n.

Fig. 5 compares AL with BASE in terms of eliminations per variable and sumptions per clause; each point is an instance, and its color indicates which strategy solved more bounds on that instance (green: AL; red: BASE; purple: tie). Both plots use a logarithmic scale. As expected, AL's operation on a smaller formula (under the current trail) yields more eliminations and, even more so, more sumptions than BASE, so that most points lie above the diagonal. However, this is not always the case, which illustrates the trade-off described in Sect. 4.2.4: BASE can eliminate variables about to be assigned by assumption propagation and also use them for selfsumption. Color analysis shows a weak correlation between the metric and the winner in the eliminations plot: above the diagonal, about 21% of instances are won by AL versus 15% by BASE, whereas below it only about 4% are won by AL compared to 10% by BASE (the rest are ties). For sumptions, the correlation is weaker.



(a) Eliminations/Vars: AL vs BASE.

(b) Sumptions/Clss: AL vs BASE.

Figure 5 AL vs BASE: (a) eliminations per variable; (b) sumptions per clause, per benchmark (log scale). Point color: green = AL solved more bounds, red = BASE, purple = tie.

6 Conclusion

We introduced *Backtrackable Inprocessing* (BI), a framework that allows inprocessing under the current trail at any level, at any point in the search, while preserving correctness under backtracking (focusing on subsumption, self-subsuming resolution, and BVE). We integrated BI into the Island solver and evaluated several strategies for incremental preprocessing on BMC benchmarks from HWMCC'17, obtained by unrolling each benchmark up to bound 100 and using a one-hour timeout. Our best BI-enhanced strategy outperforms the baseline strategy, solving ∼1.5× more bounds among those unsolved without inprocessing.

We have so far demonstrated BI's benefit in incremental preprocessing; however, BI enables a much broader design space. Inprocessing can now be integrated into the search in many ways, paving the way for new strategies and significantly more efficient SAT solving, both non-incremental and incremental.

Beyond SAT solving, BI is also promising for enabling inprocessing in state-of-the-art algorithms for model counting [19] and enumeration [20], which rely on systematic search without restarts and have therefore so far been unable to exploit inprocessing.

References

1 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. In *Handbook of Satisfiability (2nd ed.)*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.

2 Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1999)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. doi:10.1007/3-540-49059-0_14.

3 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. Cadical 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

4 Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

5 Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability, Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. doi:10.3233/FAIA336.

6 Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking competition 2017. In Daryl Stewart and Georg Weissenbacher, editors, *Formal Methods in Computer-Aided Design, FMCAD 2017, Vienna, Austria, October 02-06, 2017.*, page 9. IEEE, 2017. doi:10.23919/FMCAD.2017.8102233.

7 Aaron R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011. doi:10.1007/978-3-642-18275-4_7.

8 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005. doi:10.1007/11499107_5.

9 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando 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. doi:10.1007/978-3-540-24605-3_37.

10 Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019. doi:10.1007/978-3-030-24258-9_9.

11 Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012. doi:10.1007/978-3-642-31365-3_28.

12 Alexander Nadel. Introducing intel (r) sat solver. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.8.

13 Alexander Nadel. Artifact for "Backtrackable Inprocessing" (sat 2026). Zenodo, 2026. Dataset. doi:10.5281/zenodo.20034734.

14 Alexander Nadel. Backtrackable inprocessing, 2026. arXiv:2605.03654.

15 Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012. doi:10.1007/978-3-642-31612-8_19.

16 Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Preprocessing in incremental SAT. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 256–269. Springer, 2012. doi:10.1007/978-3-642-31612-8_20.

17 Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental SAT. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2014. doi:10.1007/978-3-319-09284-3_16.

18 João P. Marques Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *Digest of Papers: FTCS-27, The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 152–161. IEEE Computer Society, 1997. doi:10.1109/FTCS.1997.614088.

19 Mate Soos and Kuldeep S. Meel. Engineering an efficient probabilistic exact model counter. In Ruzica Piskac and Zvonimir Rakamaric, editors, *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III*, Lecture Notes in Computer Science, pages 72–91. Springer, 2025. doi:10.1007/978-3-031-98682-6_5.

20 Giuseppe Spallitta, Roberto Sebastiani, and Armin Biere. Disjoint projected enumeration for SAT and SMT without blocking clauses. *Artif. Intell.*, 345:104346, 2025. doi:10.1016/J.ARTINT.2025.104346.