

SAT-IT: An Online Interactive SAT Tracer

Wilber Bermeo¹, Jordi Coll¹, Pau Ferrer¹ and Mateu Villaret¹

¹Universitat de Girona, Girona, Spain

Abstract

Modern Boolean Satisfiability (SAT) solvers, based on the Conflict-Driven Clause Learning (CDCL) paradigm, achieve state-of-the-art efficiency but present a steep learning curve due to their sophisticated algorithms and highly optimized data structures. Understanding these complex mechanics and evaluating the effectiveness of problem encodings is notoriously challenging for students and emerging researchers. To ease this learning process, we introduce the Interactive SAT Tracer (SAT-IT), an open-access web environment designed to make the foundations of SAT solving highly visible and interactive. SAT-IT offers a staged pedagogical progression—from naive backtracking to DPLL and full CDCL with the two-watched literals scheme. Users can clearly inspect fundamental data structures, search space trails, and solving statistics. The tool’s interactive search space exploration is boosted with literal-level breakpoints for targeted inspection, alongside versatile automatic solving modes that offer both continuous real-time execution and state-based subroutine automation. Combined with a powerful “what-if” capability for stepping backward to explore alternative decisions, an instance manager, and an extensible architecture ready to support additional algorithms, SAT-IT serves as a practical, granular lens for experimenting with SAT solving algorithms and analysing encodings efficiency.

Keywords

Pedagogical tool, SAT solving tracer, algorithm visualization, interactive web-based tool

1. Introduction

Boolean Satisfiability (SAT) has established itself as a successful approach for solving hard combinatorial problems. This prominence is due both to the ease with which diverse problem domains can be reduced to SAT [1, 2, 3, 4], and the remarkable efficiency of modern SAT solvers.

This state-of-the-art efficiency is the culmination of decades of research, resulting in highly sophisticated algorithmic implementations. Modern solvers rely heavily on Conflict-Driven Clause Learning (CDCL) [5, 6, 7], augmented by highly optimized data structures and wisely used inference rules. However, the sophistication that makes these algorithms so effective also creates a steep learning curve. For newcomers, diving into the implementation details of modern SAT solvers can be a demanding task, which could be eased with a visual support.

To bridge this gap, we present SAT-IT, an open-access web environment designed to understand the foundational concepts of CDCL solving and encoding properties. The primary goal of SAT-IT is to make the mechanics of SAT solving granular and accessible, providing a visual and intuitive representation of the SAT solving process, explicitly exposing how fundamental data structures consisting of occurrence lists, watch lists, and trails, are inspected and modified during execution. Furthermore, SAT-IT exposes a set of relevant statistics of the solving process; such as the number of visited clauses, propagations, decisions, conflicts, and more. Other relevant features are discussed below.

The tool offers a staged learning experience by allowing users to interact with algorithms of increasing sophistication. Users can start with a naive backtracking algorithm, upgrade to DPLL, and advancing to a vanilla CDCL procedure with conflict analysis and lemma learning. Additionally, the Two-Watched Literals (2WL) scheme can be enabled in CDCL.

17th Pragmatics of SAT International Workshop 2026 (PoS’26)

✉ wilber.bermeo@udg.edu (W. Bermeo); jordicoll@udg.edu (J. Coll); u1978930@campus.udg.edu (P. Ferrer);

mateu.villaret@udg.edu (M. Villaret)

🌐 <https://wilberquito.github.io/> (W. Bermeo)

🆔 0009-0001-3748-8164 (W. Bermeo); 0000-0002-9385-5723 (J. Coll); 0009-0001-1244-3402 (P. Ferrer); 0000-0002-8066-3458 (M. Villaret)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

The platform maintains a detailed trace of the covered search space, listing the branches explored as trails. It provides interactive, step-by-step control over the solving algorithm’s execution flow and in subroutines such as the resolution of clauses during the conflict analysis in CDCL or the update of watches when the two-watched literal scheme is enabled. Furthermore, a literal-level breakpoint system allows users to halt the solving procedure precisely when a specific literal is assigned, enabling them to closely inspect the reasons and the consequences of that assignment and gain a deeper understanding of the solver’s behaviour and formula properties.

SAT-IT supports an automated solving procedure with two distinct modes. The first provides a continuous low-level step-by-step execution, where the UI is updated in real-time to reflect each phase of the solving process. The second is an integrated interactive mode, where the execution of subroutines involve the call of many step-by-steps automatically until a defined state is reached, and then, the UI is updated. This allows for accelerated navigation through the search space by bypassing repetitive low-level operations while maintaining granular control.

The tool supports non-linear exploration—as if it was a “*what if* analysis”—by allowing users to revert the solver until a chosen decision. This is implemented by re-simulating the preceding assignments—decisions and their corresponding propagations. Once re-simulated, users can manually select an alternative branching literal to observe how different assignments would have influenced the search trajectory.

SAT-IT supports the upload of DIMACS CNF formulas, their visualization within the interface, and their removal when no longer needed. Additionally, the system includes a collection of preloaded instances available by default.

The remainder of this paper is organized as follows. Section 2 provides a literature summary of existing tools designed to facilitate the understanding of SAT solving algorithms. Section 3 details significant data structures employed by state-of-the-art solvers and introduces our proposed visualization framework. Section 4 presents the workflow of SAT-IT by means of an example, offering insights into the various interactive functionalities. Section 5 explains the architecture that makes the SAT-IT tool unique. Finally, Section 6 presents the architectural foundations of SAT-IT and compares its relevant features with other tools.

2. Related Work

The Boolean Satisfiability Problem (SAT) is one of the most extensively studied problems in theoretical computer science. Early complete solving procedures culminated in the DPLL algorithm introduced in [8]. DPLL established the recursive backtracking framework with unit propagation (UP) that remains the conceptual basis of modern solvers.

Contemporary SAT solvers, however, are largely based on Conflict-Driven Clause Learning (CDCL), which extends DPLL with conflict analysis, non-chronological backtracking, and lemma learning. Engineering advances—specially Two-Watched Literals (2WL) for efficient propagation, activity-based branching heuristics (e.g., VSIDS), and restart strategies—have transformed SAT solving into a highly optimized industrial technology [6]. While these mechanisms are central to modern solver performance, they introduce substantial conceptual complexity compared to classical DPLL.

Despite the foundational nature of the Boolean Satisfiability problem in computer science, there has been little focus on developing tools designed to provide visual explanation of SAT solving concepts. To the best of our knowledge, DPvis is the first valuable tool for educational SAT solving, which was published in 2005 in [9]. DPvis is a Java tool to visualise the structure of SAT instances and runs of the DPLL procedure. DPvis’s main drawback is the lack of a meaningful representation of the solving process and little user-solving granularity. Latterly, in 2018 LearnSAT [10] was introduced as a pure education tool written in Prolog. The main drawback of LearnSAT is that it has no graphical support. Later on, in 2023, EduSAT [11] was introduced as a Python library that utilizes Jupyter notebooks to offer a “learning through coding” approach. EduSAT prioritizes programmatic understanding over pure abstraction, and it is constrained to DPLL solving. In the same year, the Interactive SAT Tracer

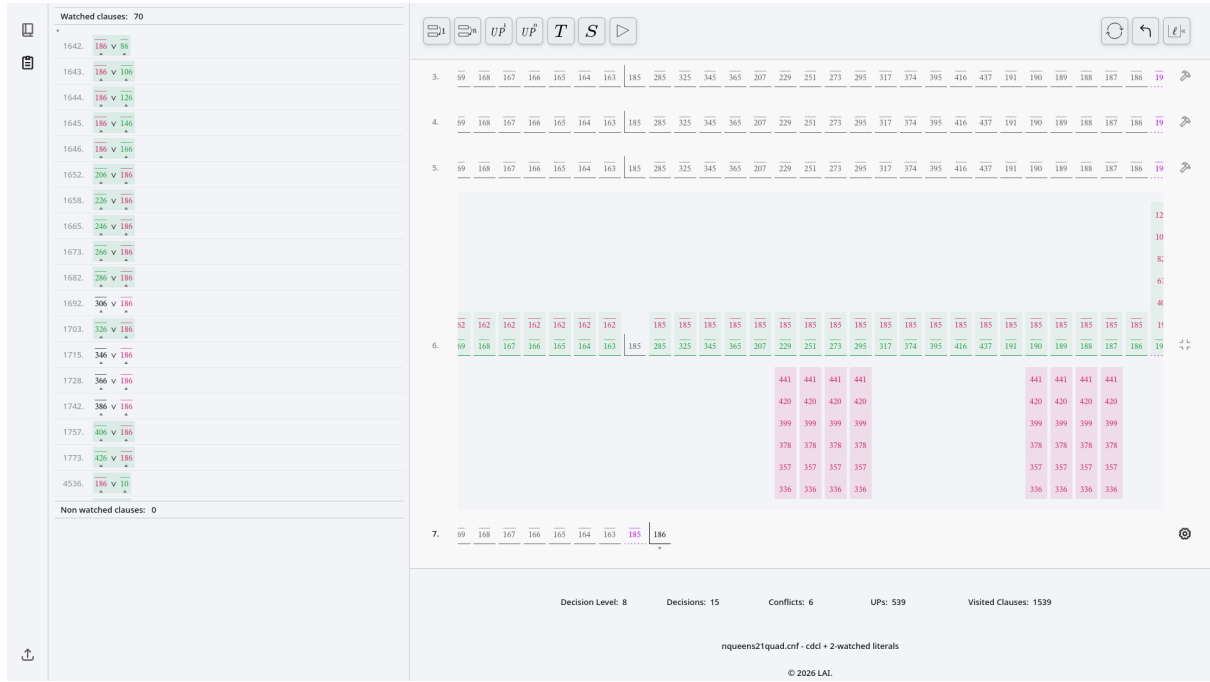


Figure 1: SAT-IT Interface Overview. The left sidebar displays the watch list for the complementary of the last assignment (186), while the top panel provides interactive controls for the solving algorithm. Bottom-row statistics track real-time metrics—including decision levels, conflicts, and visited clauses—alongside the specific algorithm and the name of the instance being solved. The central view visualizes the search space by means of trails. Each trail details branch-specific information such as conflictive and derived clauses.

(SAT-IT) [12] was introduced as a Java-based desktop application designed to illustrate algorithm execution through process tracing. Although this version incorporated the full CDCL complemented with 2WL, it lacked critical visual granularity; specifically, it provided no representation of the visited clauses within the occurrence lists during UP, nor did it offer a high-level overview of the data structures and subroutines involved in the solving process. In 2025, Madgwick introduced SAT-Web [13] as the first web-based educational tool of its kind. SAT-Web provides visualizations such as search trees and variable interaction graphs. However, SAT-Web is primarily constrained to the DPLL algorithm, leaving a pedagogical gap regarding solving algorithms. This leaves a significant opening for tools that can effectively visualize more complex SAT solving algorithms such as CDCL and its extensions.

Inspired by the efforts discussed above, we present SAT-IT (Figure 1), a web-based environment designed to bridge the gap between state-of-the-art SAT solving and pedagogical visualization. While it shares name with its Java predecessor [12], SAT-IT is not a mere reimplement. SAT-IT utilizes a modular architecture designed to support both future algorithmic extensions and diverse formalisms beyond SAT. It also allows a granular user-interaction with the implemented SAT solving algorithms. This granular capability helps users to better understand the inner behaviour of the algorithms and data structures used by the state-of-the-art solvers.

The SAT-IT source code is hosted on GitHub¹ and is publicly accessible at the project’s domain².

3. Data Structures and Visualization

SAT-IT incorporates three SAT-solving algorithms to enable a staged learning experience, allowing users to interact with solving methods of progressively increasing sophistication. The simplest algorithm is Backtracking, a naive backtracking search algorithm that iteratively branches by deciding some assignment, looks for possible falsified clauses, and prunes the branch when a clause is falsified by

¹<https://github.com/udg-lai/sat-it>

²<https://udg-lai.github.io/sat-it/>

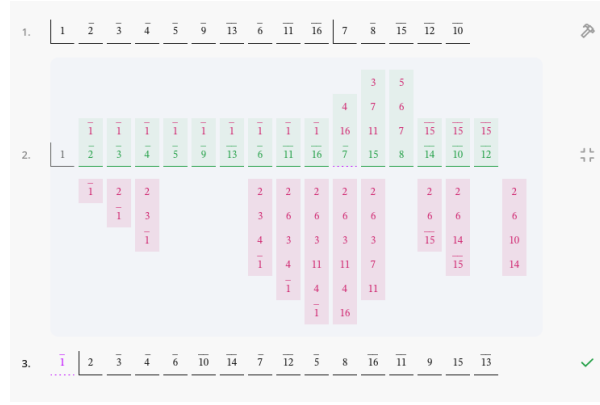


Figure 2: Search space by means of trails. Focusing on trail number two, on the top of it is possible to appreciate the reasons that propagated the literals, whereas under it the conflict analysis, with the conflictive, derived, and learned clauses are displayed from right to left.

undoing the last decision. The DPLL algorithm extends Backtracking with UP. Lastly, the CDCL algorithm extends DPLL by including conflict analysis, lemma learning, and backjumping.

In this section, we present a detailed explanation of the most important visualization mechanisms and their data structures for the three algorithms included in SAT-IT

3.1. The Trail

SAT-IT adopts the operational framework originally introduced in [8] and subsequently adapted in [12], which introduces the rule-based transition system that we use as reference for formal representation of the trail. Within this framework, a trail is a sequence of literals that represents the literals satisfied by an interpretation, ordered chronologically by assignment. This rule-based transition system enables formal reasoning about Backtracking, DPLL, and CDCL algorithms through a concise operational notation.

Within the adopted framework, the trail is transformed through the application of four rules: *Decision*, *Unit Propagation*, *Backtrack*, and *Backjump*. The *Decision* rule assigns a truth value to an unassigned literal; by default, SAT-IT assigns variables in ascending order and *truth* value.

The *Unit Propagation* (UP) rule assigns the only unassigned literal in a unit clause. A unit clause is a clause that contains exactly one unassigned literal, while all of its remaining literals are falsified. When a conflict is found, the *Backtrack* rule reverts the most recent decision. The *Backjump* rule is exclusive of CDCL, and is invoked after the *Learn* rule. To ensure a learned clause becomes unit at the earliest possible stage, the solver backjumps to the second highest decision level of the learnt clause's literals. The decision level of a literals is determined by the number of preceding decisions in the trail. Notice that the *Decision* and the *Unit Propagation* rules extend the trail, whereas the *Backjump* and *Backtrack* rules may shrink it. For an exhaustive discussion of this formal framework, the reader is referred to [12].

In SAT-IT, a trail (see Figure 2) is represented as a sequence of literals categorized by their assignment type, i.e., the rule used to assign them. A decision is denoted by a box-shaped boundary that signifies the start of a new *decision level*. Decisions might be displayed within a *collapsed* box (represented by two vertical lines and a bottom line), which hides subsequent propagations within that decision level, or by an *expanded* box (represented by a left vertical line and a bottom line), exposing all UPs associated within that level. These visual states for decided literals are interactive, allowing the user to toggle between collapsed and expanded views. Propagated literals are distinguished solely by a solid bottom line. A distinct case is also illustrated, at the beginning of the trail number three, a purple-coloured literal appear with dashed bottom lines. This visual style represents a backtracking in the context of DPLL or Backtracking algorithms. In CDCL, a literal of this kind represents the complementary of the First Unique Implication Point (1UIP) identified during *conflict analysis* in the previous trail, and hence the literal that has been propagated after backjumping [14].

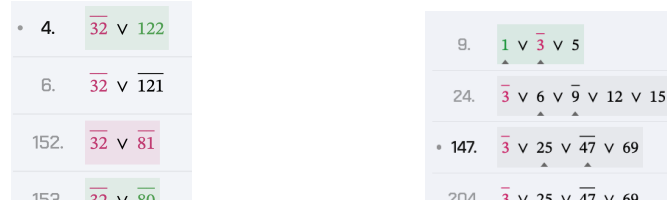


Figure 3: Visualization of data structures. Left) Occurrence list of literal $\overline{32}$. Clause 152 is currently unvisited yet already falsified. Right) Watch list of literal $\overline{3}$. Triangle indicators denote watched literals. Clauses 9, 24, and 147 have already been visited. For keeping the invariance of the 2WL scheme, the watches that pointed to literal $\overline{3}$ on clauses 24 and 147 have been updated. As a result, two non-falsified literals are watched at each clause.

As shown in Figure 2, in trail number two, a SAT-IT trail is accompanied by two more views designed to provide deeper insight of the solving procedure. The view situated on the top and spatially united to the trail shows the *antecedents*—or reasons—of propagated literals, where each column is a clause. In case the current trail does not satisfy the formula, the bottom view shows the *conflict clause* at the right of the trail. In CDCL, at the left of the conflict clause, SAT-IT shows the clauses derived during the conflict analysis. More precisely, from right to left, it shows below each propagation reason, the clause resulting from applying a resolution step between the reason and the previously derived clause located at its right. Note that the leftmost derived clause is the lemma that will be learned, and it is displayed below the UIP. In both views, clauses (reasons, conflict clause, resolvent clauses, and the derived lemma), are represented as a vertical coloured box, green (satisfied) or red (unsatisfied) depending on the formula interpretation. The colour scheme is applied consistently also to literals in the clauses. Also, both views are updated incrementally in real time as new propagated literals are added to the trail (top view) or resolution steps are applied during conflict analysis (bottom view). A trail is accompanied by a clickable button on its right to show or hide the reasons and conflict analysis steps.

SAT-IT as a visual tracer requires an effective representation of the search space. To this end, we represent the solving process of an instance F as an enumerated sequence of trails $(M_1, \dots, M_n)$ as it can be seen in Figure 2 with the number located at the left-side of each trail. In particular, when the solving process is completed, $M_i \not\models F$ for all $i < n$, and if F is satisfiable M_n is a model.

3.2. Occurrence and Watch Lists

The standard mechanism in modern solvers for applying UP while checking for conflicts is to inspect the new assigned literals added in the trail on a FIFO basis. For this purpose, a pointer or *head* specifies which literal is being inspected. For example, in Figure 3 (Left), we know that the head points to literal 32, as the occurrence list of literal $\overline{32}$ is being displayed. The pointer is moved to the right after a literal has been inspected. The inspection of a literal l consists in visiting the clauses containing its complement $\overline{l}$, since these are the clauses that might have become unit or unsatisfied after falsifying $\overline{l}$. For this purpose, SAT-IT lets the user configure the algorithm to use either watch lists (the standard and most efficient data structure) or full occurrence lists, thus enabling a comparison of performances and providing a visual motivation for the use of 2WL scheme.

The occurrence list is a visual component situated in the tool pane (see Figure 3). SAT-IT facilitates interactive navigation through this list with a pointer that indicates which clause is currently being visited. The satisfaction or unsatisfaction of the clauses in the list is visually represented employing the red/green colour-coding scheme previously established. To provide further clarity during inspection, visited clauses that are not already satisfied or unsatisfied are highlighted with a gray background. Moreover, when UP occurs due to the currently visited clause, the colour scheme of all affected clauses is updated in real time.

If 2WL scheme is enabled, instead of the occurrence list, the watch list is displayed, showing only those clauses where the complementary literal is actively being watched as shown in Figure 3 (Right). In this watch list view, specific indicators are rendered below the 2WL, which are dynamically updated as clauses are inspected and watches are updated because of the propagation procedure. Furthermore, in

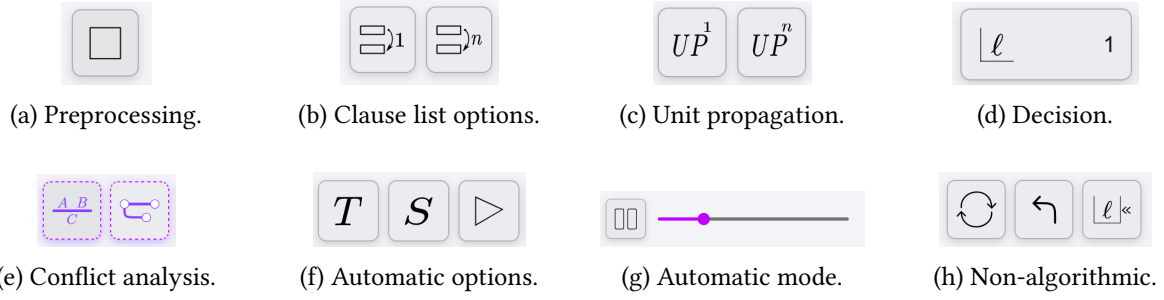


Figure 4: Relevant interactive buttons.

this specific case, the tool provides a dedicated visualization area for clauses bypassed by 2WL, offering user acknowledgment of the efficiency and behaviour of the watched literal data structures.

4. Flow Management

In this section, we highlight SAT-IT’s workflow. To this end we illustrate it by exploring a hypothetical execution in CDCL as it extends the other algorithms of the tool.

After selecting a problem, SAT-IT will activate the preprocessing phase, where a single option is available to identify all empty and unary clauses (clauses with no literals and with just one literal, respectively) (see Figure 4a). Note that this process may already determine that the formula is unsatisfiable. If this is not the case, every unary clause will add a literal to the trail by UP, and the solving process will start: the head will point to the first literal l in the trail, and the solver will transition into the *Conflict Detection* phase. This phase is in charge of inspecting the list of clauses containing $\bar{l}$ (either the occurrence list or the watched list), as detailed in Section 3.2, to find unit clauses (propagating a literal) or falsified clauses (finding a conflict). At this stage, the SAT-IT interface offers the user several inspection options in which visiting a single or all clauses from the clause list is possible (see Figure 4b).

Upon completing the clause list inspection, if no conflicts were detected, the head of the trail will point to the next literal, if any exists. To speed up the process, the SAT-IT interface offers the user the possibility to either find the next UP or execute all possible UPs (see Figure 4c).

If no conflict clauses were detected during the *Conflict Detection* phase, the state machine transitions into the *Decision* phase. By pressing the decision button (see Figure 4d) the literal displayed at the right of it will be decided and, as a consequence, the solver will enter the *Conflict Detection* phase. If a conflict is identified, the solver transitions to the *Conflict Analysis* stage. Within this stage, the user can either execute a single resolution step or automatically find and propagate the 1UIP to resolve the conflict (see Figure 4e). The recursive application of the Decision, Conflict Detection and Conflict Analysis phases, constitute the CDCL algorithm.

SAT-IT also offers the option to explore “What-if” scenarios by undoing the execution of the algorithm until a previous decision point (removing all trails after this point, and cancelling all assignments after this decision). This can be done by clicking on a decided literal and selecting the “Revert Up to Here” option from the displayed pop-up, or by pressing the *undo* button (see Figure 4h). After undoing the decision, the user can explore another search path by replacing the decision placeholder with an alternative literal.

Furthermore, SAT-IT provides an *automatic solving* functionality. This functionality might be used to finish the current trail, or to solve the entire instance and display the full searched space as a sequence of trails. Both things can be emulated by the automated *step-by-step* mode, where users can observe assignments, visited clauses, statistics, and other relevant information as they occur. The execution speed can be adjusted, and the solver can be halted at any time (see Figures 4f and 4g).

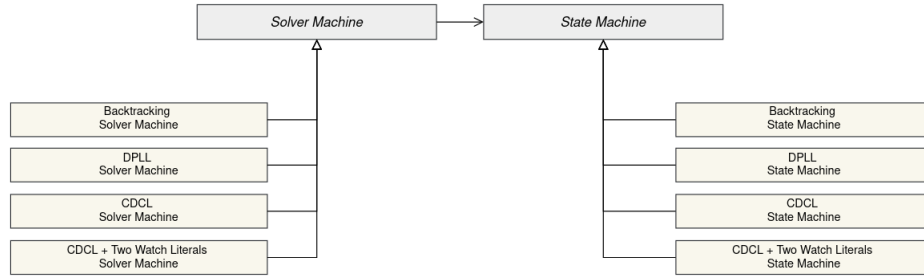


Figure 5: Solver Machine and State Machine Composition.

5. Architecture

In this section, we provide some intuitions of the software architecture upon which SAT-IT is built.

SAT-IT is a pedagogical tool that provides deep, granular insight into the internal mechanics of the implemented SAT solving algorithms by showing how data structures are updated through the solving process. To support robust user-tool interaction—including solver-generated events, atomic state inspection, and dynamic execution control—the system requires a clear separation of concerns and a high degree of decoupling between logical components. To achieve this, each SAT solving procedure in SAT-IT is modelled as a state diagram (or state machine)³, extending the rule-based framework presented in [12]. In SAT-IT, these rule-based transitions are further augmented to support user-triggered events that may alter the execution flow.

Conceptually, a solver in SAT-IT is an event-driven system that is constructed upon a *solver machine* which encapsulates a specific *state machine* (see Figure 5). The state machine defines the logical states and transition rules for a given algorithm; for instance, the CDCL state machine includes states for lemma learning and non-chronological backjumping that are absent in DPLL or Backtracking machines, though they share common operational states such as the decision phase. The solver machine acts as a controller, navigating the transitions defined by the state machine, advancing the computation, emitting events to the user interface, and reacting to user-triggered actions. All solver machines derive from a common abstract solver machine, providing a uniform API for state transitions, inspection, and event handling. This abstraction ensures system-wide consistency and facilitates the integration of new solving techniques.

6. Conclusion and Future Work

In this work, we presented SAT-IT, a specialized teaching tool for the visualization of and interaction with SAT solving processes and their underlying data structures. While SAT-IT implements standard solving algorithms, its primary objective is not high-performance benchmarking or the analysis of long-term heuristics that require extensive conflict processing. Instead, the architecture of SAT-IT is founded upon two core pillars: interactive communication and extensibility. This design allows users to dynamically monitor and influence the internal state of the solver during execution.

A solver in SAT-IT is constructed upon a *solver machine* which encapsulates a specific *state machine*. The state machine defines the logical states and transition rules for a given algorithm; for instance, the CDCL state machine includes states for lemma learning and non-chronological backjumping that are absent in DPLL or Backtracking machines, though they share common operational states such as the decision phase. The solver machine acts as a controller, navigating the transitions defined by the state machine, advancing the computation, emitting events to the user interface, and reacting to user-triggered actions. All solver machines derive from a common abstract solver machine, providing a

³A deeper explanation of the architecture and the state diagrams corresponding to the solving algorithms considered in the tool can be found in the Appendix.

Tool	Year	Platform	Primary Focus	Key Limitation
DPvis [9]	2005	Java/Desktop	Offline trace	Limited granularity interaction at solving.
LearnSAT [10]	2018	Prolog/Desktop	Offline trace	No real-time feedback.
EduSAT [11]	2023	Python/Desktop	Programmatic implementation	No real-time feedback; DPLL only.
SAT-IT [12]	2023	Java/Desktop	Interactive trace	Limited visualization.
SAT-Web [13]	2025	JS/Web	Interactive trace	DPLL only.
SAT-IT	2026	TS/Web	Interactive trace	—

Table 1
Comparison of Educational SAT Solving Tools

uniform API for state transitions, inspection, and event handling. This abstraction ensures system-wide consistency and facilitates the integration of new solving techniques.

Currently, SAT-IT is implemented using the *Svelte* [15] web framework and *TypeScript* [16] to ensure type safety and system robustness. The user interface leverages *KaTeX* [17] for high-quality mathematical rendering, while *GitHub Actions* are employed to facilitate continuous integration and ensure global accessibility.

As illustrated in Table 1, SAT-IT differs from existing tools by providing richer solving exploration environment and novel features. Future development of SAT-IT will focus on incorporating advanced SAT techniques, such as lemma shortening [18], chronological backtracking in CDCL [19] and visual implication graphs for analysing conflicts, as well as extending the framework to related paradigms like MaxSAT and Pseudo-Boolean (PB) solving.

Finally, SAT-IT intends to be the main playground for exploring, analysing and understanding SAT-based algorithms for newcomers. Future iterations will facilitate the exchange of pedagogical resources, encouraging a collaborative ecosystem driven by the community. This functionality will be of particular significance to educators, enabling the sharing of interactive, guided exercises and illustrative examples that significantly enhance the learning tour.

Acknowledgments

This work was funded by MCIN/MICIU/AEI/10.13039/501100011033 and by ERDF A way of making Europe (grants PID2021-122274OB-I00 and PID2024-157625OB-I00).

Declaration on Generative AI

During the preparation of this work, the authors used Gemini in order to: Paraphrase and reword, Improve writing style, and Grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and they take full responsibility for the publication’s content.

References

- [1] D. Kaufmann, A. Biere, M. Kauers, Verifying large multipliers by combining sat and computer algebra, in: 2019 Formal Methods in Computer Aided Design (FMCAD), IEEE, 2019, pp. 28–36.
- [2] H. A. Kautz, B. Selman, et al., Planning as satisfiability., in: ECAI, volume 92, 1992, pp. 359–363.
- [3] E. Demirović, N. Musliu, F. Winter, Modeling and solving staff scheduling with partial weighted maxsat, *Annals of Operations Research* 275 (2019) 79–99.

- [4] M. Bofill, J. Coll, M. Garcia, J. Giráldez-Cru, G. Pesant, J. Suy, M. Villaret, Constraint solving approaches to the business-to-business meeting scheduling problem, *Journal of Artificial Intelligence Research* 74 (2022) 263–301.
- [5] J. P. Marques-Silva, K. A. Sakallah, Grasp: A search algorithm for propositional satisfiability, *IEEE Transactions on computers* 48 (2002) 506–521.
- [6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient sat solver, in: *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 530–535.
- [7] J. Marques-Silva, I. Lynce, S. Malik, Conflict-driven clause learning sat solvers, *Handbook of satisfiability* (2009) 131–153.
- [8] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t), *Journal of the ACM (JACM)* 53 (2006) 937–977.
- [9] C. Sinz, E.-M. Dieringer, Dpvis—a tool to visualize the structure of sat instances, in: *International Conference on Theory and Applications of Satisfiability Testing*, Springer, 2005, pp. 257–268.
- [10] M. Ben-Ari, Learnsat: A sat solver for education, *Journal of Open Source Software* 3 (2018) 639. URL: <https://doi.org/10.21105/joss.00639>. doi:10.21105/joss.00639.
- [11] Y. Zhao, Z. An, M. Ma, T. Johnson, Edusat: A pedagogical tool for theory and applications of boolean satisfiability, 2023. URL: <https://arxiv.org/abs/2308.07890>. arXiv:2308.07890.
- [12] M. Cané, J. Coll, M. Rojo, M. Villaret, SAT-IT: The Interactive SAT Tracer, in: *Proceedings of the 25th International Conference of the Catalan Association for Artificial Intelligence, CCIA 2023*, 2023, pp. 337–346. doi:10.3233/FAIA230704.
- [13] J. Madgwick, M. Lester, Sat-web: a web-based educational sat visualisation tool, in: *Pragmatics of SAT 2025*, 2025.
- [14] J. M. Silva, K. A. Sakallah, Grasp-a new search algorithm for satisfiability, in: *Proceedings of International Conference on Computer Aided Design*, IEEE, 1996, pp. 220–227.
- [15] Svelte contributors, Svelte, 2023. URL: <https://svelte.dev>.
- [16] Microsoft Corporation, Typescript: Javascript with syntax for types, <https://github.com/microsoft/TypeScript>, 2024.
- [17] KaTeX contributors, Katex: Fast math typesetting for the web, <https://github.com/KaTeX/KaTeX>, 2024.
- [18] N. Sörensson, A. Biere, Minimizing learned clauses, in: *International Conference on Theory and Applications of Satisfiability Testing*, Springer, 2009, pp. 237–243.
- [19] S. Möhle, A. Biere, Backing backtracking, in: *International Conference on Theory and Applications of Satisfiability Testing*, Springer, 2019, pp. 250–266.

A. Online Resources

The online resources of the project can be found on:

- Source Code
- SAT-IT Website

B. Architecture

B.1. State Machine

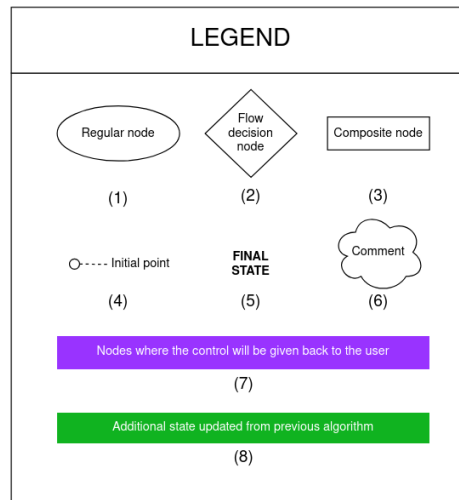


Figure 6: Legend of SAT-IT's State Machines.

The deterministic nature of all SAT-solving algorithms implemented in SAT-IT allows them to be modelled as state machines. Moreover, the requirement for user-interactions to control their execution flow, motivates to represent these algorithms as state machines rather than as standard procedures. This design choice facilitates a smooth progression from Backtracking to DPLL and ultimately to CDCL and its extension with two watch literals, while preserving the interactive nature of the system.

Figures 7, 8, 9 and 10 illustrate the state machine of the solving algorithms implemented in SAT-IT. To facilitate a clearer understanding, we explain the conceptual components that may appear in the state diagrams using the legend illustrated on Figure 6.

- 1) Regular node:** An atomic unit within a solving procedure.
- 2) Flow decision node:** A branching point in the state machine's execution flow.
- 3) Composite node:** A high-level abstraction of the solving process that encapsulates multiple regular and decision nodes.
- 4) Initial point:** The entry node marking the beginning of the state machine execution block.
- 5) Final state:** The termination point of the process, reached once a model is identified or the problem is proven unsatisfiable.
- 6) Comment:** An annotation used to provide conceptual clarification for specific components.
- 7) Violet border nodes:** Specialized regular or decision nodes that interrupt the standard execution flow to return control to the user.

- 8) **Green border nodes:** Regular or decision nodes newly introduced by the algorithm. For instance, the CDCL algorithm introduces the composite node that contains the conflict analysis and backjumping that DPLL does not have.

B.2. Solver Machine

The solver machine acts as a controller between the user-interaction through the application and the solving procedures. This design allows each solver to define and manage additional parameters or logic unique to its solving algorithm, supporting a modular and extensible architecture.

The primary concepts that any solver machine grasps are:

transition: Moves from one state to another of the state machine.

step: Encapsulates the particularity of visiting the state machine of each solver. This is a logic step and typically makes multiple transitions over its state machine.

step-by-step: Implemented as higher-order-function. Encapsulates the execution of many logic steps until a certain halt condition is accomplished.

transition-by-event: Encapsulates the particularity of visiting the state machine of each solver by running a single step or many of them depending on the event sent from the application because of a user interaction.

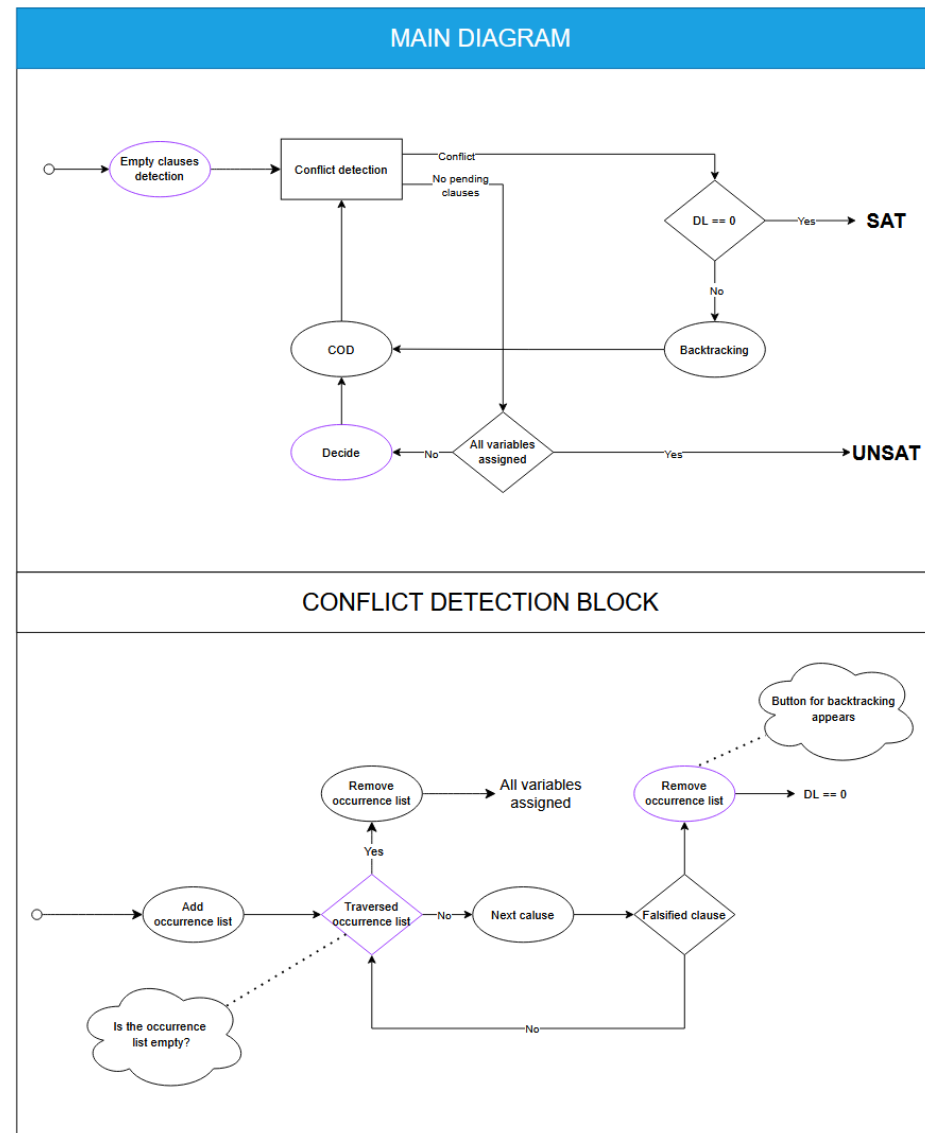


Figure 7: Backtracking state machine diagram

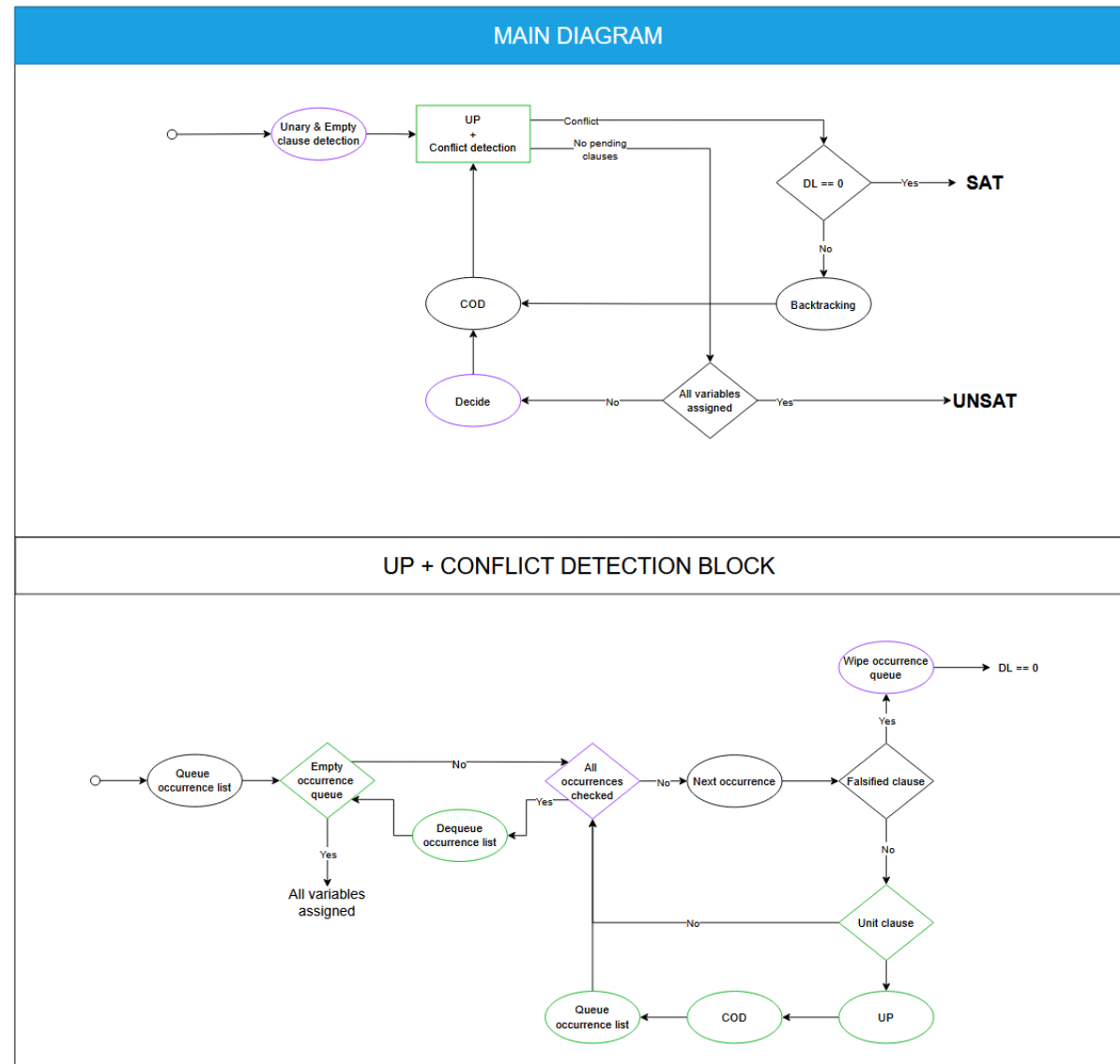


Figure 8: DPLL state machine diagram

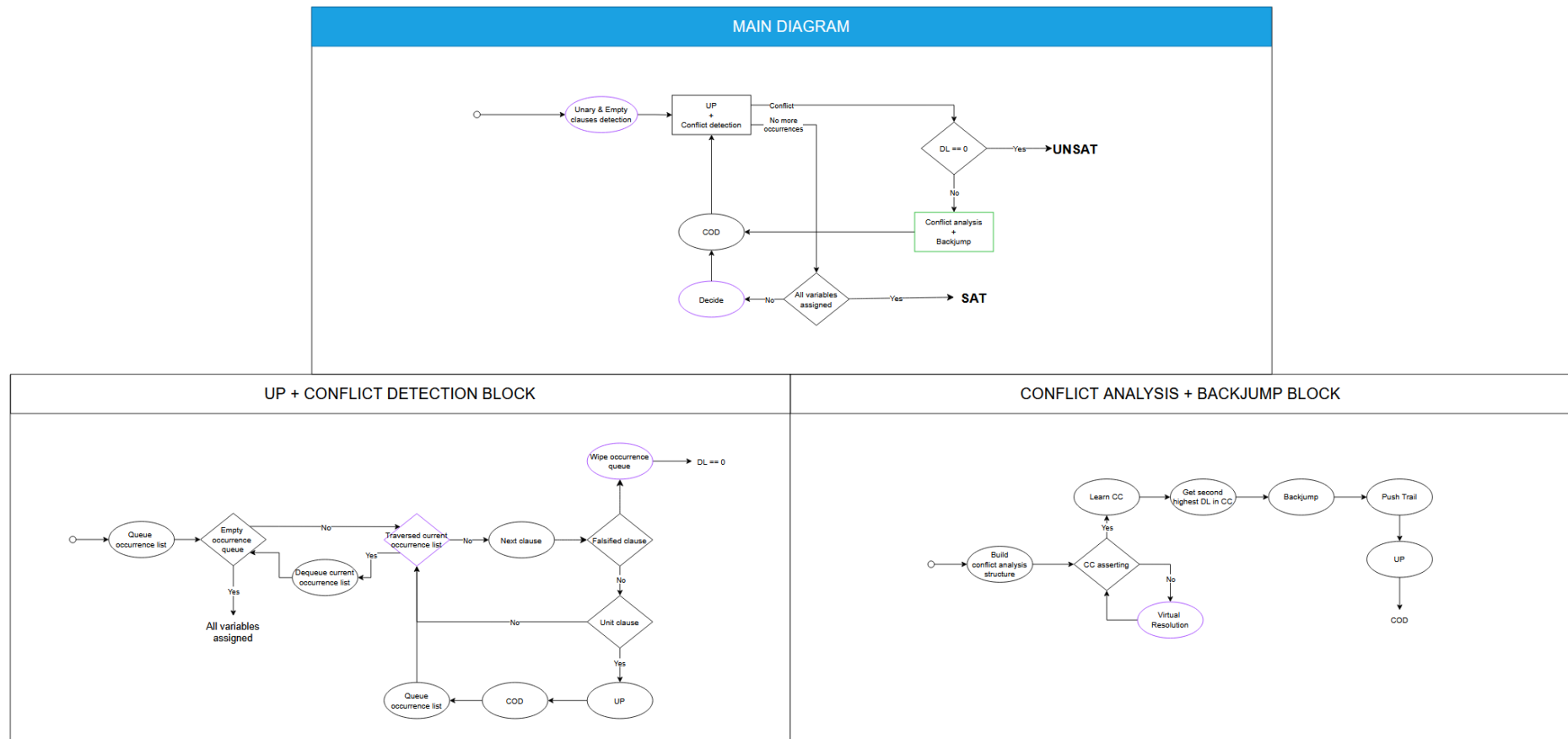


Figure 9: CDCL state machine diagram

