

Undergraduate/Master's Thesis

Model-Based API Fuzzing for the Java SAT Solving Library SAT4J

Iris Parruca

Examiner: Prof. Dr. Armin Biere

Advisers: Dr. Mathias Fleury, Tobias Paxian

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair of Computer Architecture

September 10th, 2024

Writing Period

25. 04. 2024 – 10. 09. 2024

Examiner

Prof. Dr. Armin Biere

Second Examiner

Prof. Dr. Daniel Le Berre

Advisers

Dr. Mathias Fleury, Tobias Paxian

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, 10.09.2024

Place, Date

A handwritten signature in black ink, consisting of stylized, overlapping letters, positioned above a horizontal line.

Signature

Abstract

Building trust in SAT solvers is crucial for their effective application in complex problem-solving tasks. Model-based API fuzzing has been demonstrated to be effective in identifying issues in SAT and SMT solvers. Additionally, recent developments have introduced a proof system for incremental calls, though this technology has thus far been implemented only in C++ solvers. This thesis focuses on implementing these advancements in Sat4j, the only competitive SAT solver available in Java. Despite Sat4j’s extensive testing over the years, integrating model-based API fuzzing and incremental proof-checking proved to be a substantial enhancement. The tool revealed five previously undetected bugs ranging from “Null Pointer Exceptions” and “Assertion Errors” to wrong models returned by the solver. This work improves the robustness of Sat4j and sets a new benchmark for Java-based SAT solvers.

Contents

1	Introduction	1
1.1	Overview	3
2	Background	5
2.1	SAT Solving	5
2.1.1	CDCL	7
2.1.2	Data Structures	9
2.1.3	Branching Heuristics	9
2.1.4	Restarts	10
2.1.5	Proofs	12
2.2	SAT4J	13
2.3	Fuzzing	17
2.3.1	Random Generators in Java	17
2.4	Delta Debugging	18
3	Related Work	21
3.1	API Fuzz Testing	21
3.2	Incremental Proof-Checking	23
4	Workflow Implementation in Sat4j	27
4.1	Test Case Generator	27
4.1.1	Option Fuzzing	28
4.1.2	Clause Generation	33

4.1.3	Assumptions	35
4.1.4	Enumeration Fuzzing	35
4.1.5	Incremental Solving	36
4.1.6	Integrating IDRUP-CHECK	38
4.1.7	Statistics	40
4.2	Trace Interpreter	41
4.3	Delta Debugger	41
5	Results	45
5.1	Code Coverage	45
5.2	Bugs Identified from API Fuzzing	50
5.2.1	PureOrder Heuristics	50
5.2.2	MinOne Solver	51
5.2.3	Internal Enumerator	52
5.2.4	Head/Tail Data Structure	53
5.3	Bugs Identified from Proof-Checking	54
5.3.1	DB Simplification	55
5.3.2	ExpensiveSimplificationWLOnly Learned Clauses	56
5.3.3	MinOne Solver	57
6	Conclusions	59
7	Acknowledgments	61
	Bibliography	67

List of Figures

1	DIMACS and iCNF	6
2	PicoSAT Restart Strategy	11
3	DRUP and IDRUP	14
4	SAT4J Features	15
5	Fuzzing and Delta Debugging	19
6	Fuzzing and Delta Debugging Extended Workflow	22
7	IDRUP-CHECK State Machine	26
8	Data Structure Factory Option Constraints	30
9	Simplification Option Constraints	31
10	Partial IDRUP Proof from Parallel Solver	39
11	Trace and Reduced Trace	43
12	SolverFactory Coverage	46
13	Data Structures Coverage	47
14	Order and Phase Heuristics Coverage	47
15	Learning Strategies Coverage	48
16	Restart Strategies Coverage	48
17	Internal Enumerator Coverage	49
18	External Enumerator Coverage	49
19	Trace that identified the PureOrder heuristic bug	50
20	Trace that identified the MinOne solver bug	51

21	Trace that helped narrow down the internal enumerator bug	52
22	Trace that identified the HT data structure bug	53
23	Trace that identified the ignored clauses when generating IDRUP proof	54
24	Trace that identified the DB simplification bug	55
25	Trace that identified the DB simplification bug when enumerating . .	56
26	Trace that identified the expensiveSimplificationWLOnly bug	57

List of Algorithms

1	CDCL algorithm	8
2	Option Fuzzing Algorithm	29
3	Clause Generation Algorithm	34
4	Incremental Solving Algorithm	37
5	TraceRunner Algorithm	42
6	Extended Delta Debugging Algorithm	44

1 Introduction

The propositional satisfiability problem (SAT) is the problem of deciding if there is a truth assignment under which a given propositional formula evaluates to true. It is the classic NP-complete problem and has attracted much attention from researchers [1]. The performance improvements made to SAT solvers since the mid-90s motivated their usage to a wide range of practical applications. In some cases, SAT provided remarkable performance improvements. Successful examples of practical applications of SAT include software model checking, software testing, package management in software distributions, test-pattern generation in digital systems, identification of functional dependencies in Boolean functions, and circuit delay computation [2].

Such applications are rarely limited to solving just one decision problem but will solve a sequence of related problems. Modern SAT solvers handle such problem sequences through their incremental SAT interface. This interface allows the solver to reuse information across several related consecutive problems and avoid repeatedly loading common subformulas [3]. The resulting performance improvements make incremental SAT a crucial feature for SAT solvers in real-life applications [3].

Since SAT solvers are used in sensitive applications their misbehavior could be financially expensive and dangerous from a security perspective. A trusted SAT-solving system is vital. Many testing techniques exist to achieve reliability in SAT solvers such as unit testing, regression testing, fuzzing, and more. Fuzzing started as a way of detecting security bugs but has since progressed to include both blackbox and whitebox testing techniques [4].

The SAT solving competition is an annual event that aims to identify new challenging benchmarks, promote new solvers for the propositional satisfiability problem, and compare them with state-of-the-art solvers [5]. Brummayer et al. [6] fuzzed solvers that participated in the 2009 SAT solving competition and found three defective solvers, including critical defects causing segmentation faults in the winner of the MiniSat hack track. They also found non-deterministic crashes in the winner of the parallel solver application track and revealed that the best and second-best solvers in the random track sometimes generate invalid models. This closely relates to Brummayer and Biere [7] where they demonstrate that many critical defects can be found in SMT solvers via grammar-based black-box fuzz testing and propose to complement traditional testing approaches with this technique.

Proof-checking is another approach for achieving a higher level of confidence in SAT solvers' results. A proof-checker can be used to validate each outcome of the solver independently. This requires the solver to produce not only SAT or UNSAT answers but also a proof trace for UNSAT instances that is thought of as a certificate justifying its outcome [8]. The most common proof format for SAT solvers is based on RUP (Reverse Unit Propagation) [9]. A new proof system has been adapted for incremental solving by Fazekas et al. [10] called IDRUP that can be checked by their proof-checker IDRUP-CHECK [11]. Their experiments show that the new format is practical and has the potential to support efficient certification and verification of incremental use cases of modern SAT solvers.

These testing and verification frameworks have so far, to my knowledge, only been implemented in C++. This project aims to increase the reliability of Sat4j [12], an open-source library of SAT solvers in Java, by adding a model-based API fuzzing framework with proof-checking of the generated inputs and proofs via an external proof-checker. The library has been adopted by various Java-based academic software, in software engineering, and the popular Eclipse open platform [13]. Sat4j has been heavily used and tested through the years so very few bugs remain.

1.1 Overview

The thesis first introduces the necessary background information on SAT solving, explains what fuzzing and delta debugging are, and the motivation behind using them (Chapter 2). It also provides an overview of Sat4j and its capabilities (Chapter 2) and an overview of existing implementations of API fuzzing frameworks and incremental proof-checking (Chapter 3). After covering the theoretical information the thesis details the implementation process of adding model-based API fuzzing along with proof-checking to Sat4j which as far as I know has not been done before (Chapter 4) and then presents the bugs found as a result of this work (Chapter 5). The thesis concludes by acknowledging its limitations and the potential future works that can be conducted to extend its capabilities (Chapter 6).

2 Background

This chapter introduces the necessary knowledge on SAT solving to properly understand the design choices made when implementing the API fuzzing tool. It presents Sat4j by providing an overview of its capabilities and configurations. It then goes on to explain what fuzzing is and why delta debugging is an important addition to any tool that uses it.

2.1 SAT Solving

Propositional satisfiability (SAT) is the problem of finding a satisfying truth assignment for a given propositional logic formula or determining that no such assignment exists. SAT solvers classify the formula as satisfiable or unsatisfiable through a decision procedure. Most solvers accept an input formula in Conjunctive Normal Form (CNF). Conjunctive Normal Form is a conjunction of clauses where each clause is a disjunction of literals and each literal is either a plain or a negated variable. Figure 1 contains an example of a CNF formula in the so-called DIMACS file format [14]. It has a header *p cnf V C* that states the number of variables (V) and the number of clauses (C) needed to properly parse the input file, followed by every clause of the formula. Clauses are represented as a list of their literals and end with a 0 meaning that literals can only be encoded as nonzero integers.

As mentioned in Chapter 1, SAT solvers have found many industrial applications, and such applications are rarely limited to solving just one decision problem. A single

application will typically solve a sequence of related problems. A format called iCNF (incremental CNF) was introduced by Wieringa et al. [15] where one could describe a fixed set of input clauses and multiple sets of assumptions in a single file.

Fazekas et al. [10] refined the initial iCNF model and extended it to describe every formula-related interaction between the user and the solver, capturing arbitrary incremental problem sequences. Figure 1 shows an example of an iCNF file. The header line of iCNF files consists of *p icnf*, notably without the exact number of variables or clauses of the problem as they are unknown. Input clauses *i* represent a clause addition step and are expressed in DIMACS format. Query *q* indicates that the solver is asked to solve the current formula under the defined (possibly empty) set of assumed literals. Status *s* records the answer of the SAT solver for the previous query. Model *m* represents the assignments of all the variables the SAT solver returned after finding a satisfactory answer. Unsatisfiable cores *u* list all the failed assumptions of the last query when the SAT solver returns an unsatisfiable answer. Since iCNF is meant to describe the interaction between users and SAT solvers, the order of the commands is very important [10].

DIMACS	iCNF
p cnf 4 3	p icnf
1 2 0	i 1 2 0
2 3 -4 0	q 0
-3 1 0	s SATISFIABLE
	m 1 2 0
	i 2 3 -4 0
	i -3 1 0
	q -1 2 3 0
	s UNSATISFIABLE
	u -1 3 0

Figure 1: DIMACS and iCNF file examples [10].

A unit clause is a clause of length one and it forces its only literal to be true. Unit propagation is an important technique used in SAT solvers and works as follows: Given a formula F , repeat the following until fixpoint: If F contains a unit clause l , remove all clauses containing l and remove all literal occurrences of \bar{l} . If unit propagation on a formula F produces an unsatisfiable empty clause (i.e. conflict), F is unsatisfiable.

2.1.1 CDCL

The leading paradigm for solving satisfiability problems is the conflict-driven clause learning (CDCL) approach. CDCL solvers are primarily inspired by DPLL (Davis – Putnam – Logemann – Loveland) solvers [16]. DPLL corresponds to backtrack search, where at each step a variable x and a propositional value, either 0 or 1, are selected for branching purposes [17]. The logical consequences of each branching step are evaluated and each time an unsatisfied clause (i.e. a conflict) is identified, backtracking is executed [17]. Backtracking undoes branching steps until an unflipped branch is reached. When both values have been assigned to the variable, backtracking will undo this branching step [17]. If both values have been considered for the first branching step and resulted in conflicts, then the CNF formula can be declared unsatisfiable [17].

Algorithm 1 shows the standard organization of a CDCL SAT solver, which follows the organization of DPLL, the main differences being the call to function *ConflictAnalysis* each time a conflict is identified, and the call to *Backtrack* when backtracking takes place [16]. Marques-Silva et al. [16] explain how each time the CDCL SAT solver identifies a conflict due to unit propagation the *ConflictAnalysis* procedure is invoked where one or more new clauses are learnt, and a backtracking decision level is computed. The conflict analysis procedure analyzes the structure of unit propagation and decides which literals to include in the learnt clause. *Backtrack* backtracks to the decision level computed by *ConflictAnalysis*.

Unrestricted clause recording can be impractical. Learnt clauses consume memory and large clauses are known for being useless due to the added overhead [16].

Algorithm 1 CDCL algorithm

```

CDCL ( $\varphi, \nu$ )
  if (UnitPropagation( $\varphi, \nu$ ) == CONFLICT) then
    return UNSAT
  end if
   $dl \leftarrow 0$  ▷ Decision level
  while not AllVariablesAssigned( $\varphi, \nu$ ) do
    ( $x, v$ ) = PickBranchingVariable( $\varphi, \nu$ ) ▷ Decide stage
     $dl \leftarrow dl + 1$  ▷ Increment decision level due
    to new decision
     $\nu \leftarrow \nu \cup \{(x, v)\}$ 
    if UnitPropagation( $\varphi, \nu$ ) == CONFLICT then
       $\beta = \text{ConflictAnalysis}(\varphi, \nu)$  ▷ Diagnose stage
      if  $\beta < 0$  then
        return UNSAT
      else
        Backtrack( $\varphi, \nu, \beta$ )
         $dl \leftarrow \beta$ 
      end if
    end if
  end while
  return SAT

```

Glucose, proposed by Audemard and Simon [18], pioneered the state-of-the-art approach by defining LBD (Literal Blocks Distance) that measures the number of distinct decision levels in a given learnt clause. The lower the LBD score of a learnt clause, the better its quality. With this score in mind, they build an aggressive cleaning strategy.

2.1.2 Data Structures

A key aspect of SAT solvers is the data structure used for storing clauses, variables, and literals. The implemented data structure dictates how unit propagation and conflict analysis are implemented and significantly impact the SAT solver's run time performance [16].

Two data structures proposed for SAT, presented by Marques-Silva et al. [16], are the Head and Tail (H/T) and the Watched Literals (WL) data structures. The H/T data structure associates two references with each clause, the head (H) and the tail (T). Each time a literal pointed to by either reference is assigned, a new unassigned literal is searched for. In case an unassigned literal is identified, it becomes the new reference. If no unassigned literal can be identified, then the clause is declared, unsatisfied or satisfied, depending on the value of the literal pointed to by the other reference.

With WL, as with H/T, two references are associated with each clause, however, there is no order relation between the two references, allowing the references to move freely. This has the key advantage that no literal references need to be updated when backtracking. In contrast, a clear drawback is that unit or unsatisfied clauses are identified only after traversing all the clauses' literals.

2.1.3 Branching Heuristics

One key aspect of SAT algorithms is how assignments are selected at each algorithm step, i.e. the branching heuristics [19]. Branching heuristics can be classified as static or dynamic. Static heuristics determine one linear order of assignment before the solver is started and are typically quite fast. Dynamic heuristics use information on the state of the search to make the next decision and are used by most modern SAT solvers.

VSIDS (Variable State Independent Decaying Sum) is a well-known branching heuristic with low computational overhead. As stated by Liang et al. [20] VSIDS ranks all the formula variables while running the solver. The key idea is to collect statistics over learnt clauses to guide the direction of the search. VSIDS assigns a number, called activity, to each variable in the formula and when the solver learns a clause, a set of variables is chosen, and their activities are additively increased (bumped). At regular intervals, the activities of all variables are multiplied by a constant α called the decay factor. VSIDS picks the variable with the highest activity to branch on.

Phase heuristics are part of decision heuristics but instead of choosing the variable, they choose the variable’s assignment. An approach used in Sat4j is RSAT phase saving as presented by Pipatsrisawat and Darwiche [21]. Every time the solver performs a backtrack and erases assignments, each erased assignment is saved. Any time the solver decides to branch on variable v it uses the saved assignment if one exists, otherwise, the solver uses another default phase selection heuristic.

2.1.4 Restarts

Sometimes it may be a good strategy to abandon the search and start from the beginning. For satisfiable instances, the solver may get stuck in the unsatisfiable part while for unsatisfiable instances focusing on one part might miss short proofs. One of the key ingredients of modern SAT solvers are efficient restarting schemes [22]. Randomization and learnt clauses help to avoid running into the same dead end.

As presented by Biere and Fröhlich [22] restarts can be categorized into static and dynamic. In a static restart strategy, the solver performs a restart exactly after a certain r number of conflicts. Uniform restart intervals have r as a fixed constant. Non-uniform restart intervals have r as a function of the number of restarts that have already been performed and can be realized by implementing a geometric policy, or based on the Luby series (Equation (1)).

$$luby(i) := \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1 \\ luby(i - 2^{k-1} + 1) & \text{if } 2^{k-1} \leq i \leq 2^k - 1 \end{cases}, \text{ for some } k \in \mathbb{N} \quad (1)$$

Luby Series [22]

Biere [23] presents an aggressive nested restart scheme, implemented by PicoSAT which can be seen in Figure 2 which triggers fast restarts with a high frequency. To avoid repeatedly revisiting the same search space the last learned clause before a restart is fixed and never deleted. Other learned clauses are garbage collected in the reduction phase based on their activity.

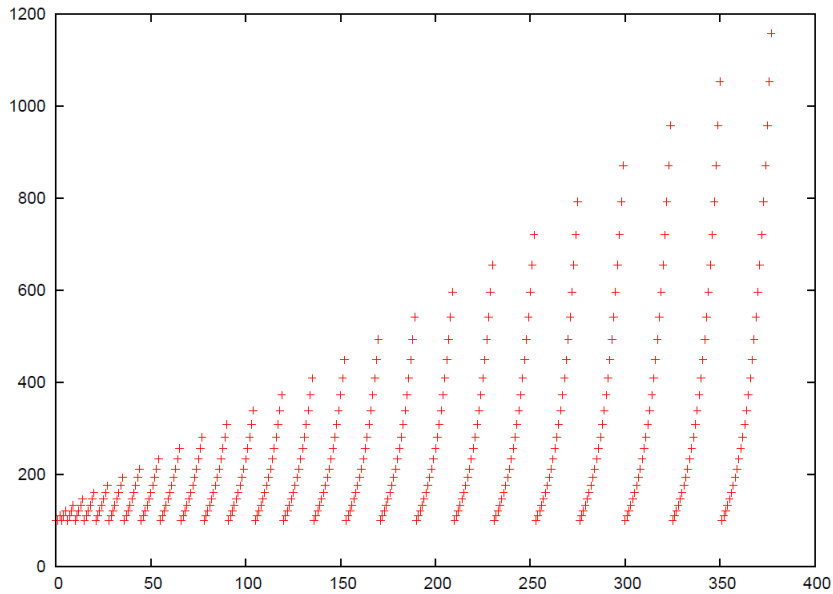


Figure 2: On the horizontal axis, the number of restarts is shown while the vertical axis denotes the number of conflicts of the restart interval, which more precisely is the number of conflicts that have to occur before the next restart is triggered. The last restart in this schedule occurs after 104408 conflicts [23].

Glucose is a dynamic restart strategy where restarts are based on the literal block distance (LBD) of learned clauses (Section 2.1.1). Clauses with a small LBD are more important in solving a SAT formula. Due to this, a search in Glucose continues as long as the set of recently learned clauses seems to be “good” according to LBD [24].

2.1.5 Proofs

A proof-generating SAT solver produces additional evidence to support its claims. For SAT claims the certificate is simply an assignment, i.e. an enumeration of Boolean variables to be set to true which is verified by substituting the Boolean values into the formula [8]. For UNSAT claims, the solvers return a proof trace as a certificate [8]. Proof logging of unsatisfiability claims is based on two approaches: resolution proofs and clausal proofs [25]. Resolution proofs require for learnt clauses (lemmas) a list of antecedents [25]. On the other hand for clausal proofs, the proof-checker needs to find the antecedents for the lemmas, consequently, resolution proofs are easier and faster to validate than clausal proofs [25].

The most common proof format is based on RUP (Reverse Unit Propagation). As presented by Van Gelder [9] RUP proofs are a sequence of clauses where each clause C_i is either an input clause or can be derived from a single RUP inference step. A clause C has RUP w.r.t. a formula F (i.e., can be derived from F with a RUP inference step) if unit propagation on the conjunction of F and the negation of C leads to a conflict. Furthermore, Fazekas et al. [10] state that inference steps can be used to add and remove clauses from formulas. When a clause can be derived from other clauses, it can be removed from the problem without changing the satisfiability of F . When SAT solvers include such deletion steps the proof is called DRUP. In DRUP proofs any clause can be deleted, not just learnt clauses. An example of a DRUP proof can be found at Figure 3.

Fazekas et al. [10] proposed an incremental proof format IDRUP which compliments iCNF in the same way DRUP format compliments DIMACS by listing learnt clauses and deleted clauses. A major difference is that all the original input clauses must also be repeated in the proof. In this regard, the iCNF interactions file forms a subsequence of the IDRUP proof. The header of this format is *p idrup*, input clauses, SAT queries, solver answers, models, and unsatisfiable cores are represented the same way as in iCNF (with tags *i*, *q*, *s*, *m*, and *u*). These lines are essential to synchronize the incremental SAT queries and the proof. Figure 3 presents an example of an IDRUP proof.

Fazekas et al. [10] introduced additional tags to capture the formula manipulation steps of the solver. An incremental proof describes which clauses were added to and removed from the formula with precision. The format supports two types of clause deletion: drop (indicated by the prefix *d*) and weaken (tagged with *w*). Dropped clauses are eliminated from the formula. Weakened clauses are eliminated from the formula as well but might be reintroduced in later steps, thus their deletion is seen as temporary. Weakened clauses were motivated by the fact that some simplification steps could prove to be invalid and must be undone. The clauses indicated by the prefix *l* are added to the problem because they are derived and then learnt by the solver. Some clauses are reintroduced via a restore step (marked with tag *r*), i.e. by a step that undoes a previous weakening step of the solver.

2.2 SAT4J

Sat4j is an open-source library of SAT solvers that aims to enable Java programmers to access cross-platform SAT-based solvers [26]. It has been adopted by various Java-based academic software, in software engineering, bioinformatics, formal verification, and the popular Eclipse open platform [13]. The Sat4j library started in 2004 as an implementation of the Minisat specification in Java [13].

DRUP	IDRUP
1 2 0	p idrup
d 1 2 -3 0	i -1 3 4 0
1 0	i -1 3 -4 0
d 1 2 0	i -2 -3 4 0
d 1 3 4 0	i -2 -3 -4 0
d 1 -2 -4 0	q 1 2 0
2 0	l -4 -2 -1 0
0	l -2 -1 0
	d -4 -2 -1 0
	s UNSATISFIABLE
	u 2 1 0
	i -1 2 0
	i 1 -2 0
	q 0
	s SATISFIABLE
	m -1 -2 -3 -4 0
	i 1 2 0
	q 0
	l 2 0
	l -1 0
	l 0
	s UNSATISFIABLE
	u 0

Figure 3: DRUP and IDRUP format examples [10].

Sat4j allows solving a range of decision and optimization problems using pseudo-boolean solving [13, 12]. It has been developed to allow testing various combinations of features developed in new SAT solvers while keeping the technology easily accessible to a newcomer [13]. Figure 4 shows an overview of features available in Sat4j and the ones encapsulated inside the red square are what this project builds upon.

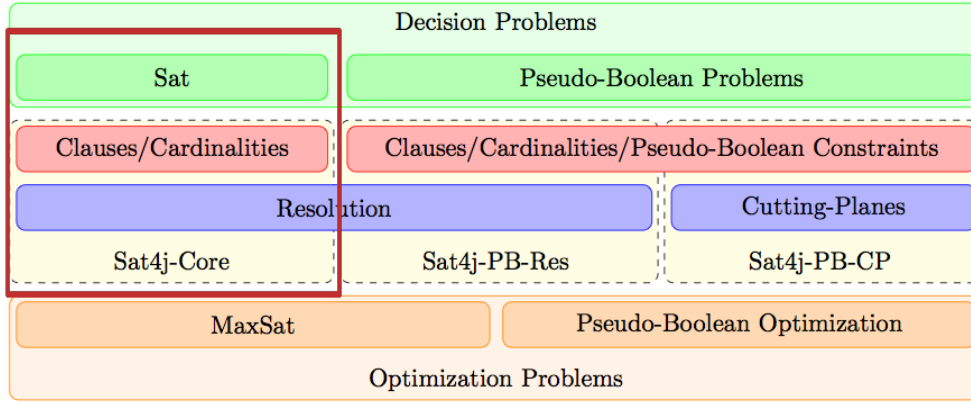


Figure 4: Overview of the features available in the SAT4J library [13].

As stated by Le Berre and Rousse [27] Sat4j provides a wide range of prebuilt solvers. The configuration that performs the best on a wide range of benchmarks from the application category of the SAT competition, *Glucose21*, is set as the default solver. The settings used in the default SAT solver include the rapid restarts strategy [23], the conflict clause minimization of Minisat 1.14 [28], the lightweight caching scheme for phase selection of RSAT [21], and finally, the solver keeps derived clauses with literals from few different decision levels as proposed in 2009 award winner Glucose [18].

However, a different configuration could perform much better on a specific class of benchmarks. This led to most of the key components of the solver being made configurable to allow the end users of Sat4j to construct a solver for their specific problem [27]. Such an approach is based on both the ability to dynamically change the main strategies of the solver and display to the user several metrics that inform about the state of the solver [27]. This way, prebuilt solvers are useful to record a combination of features that have been proven useful [27].

Le Berre and Rousse [27] provide a list of the on-the-fly solver configurations:

- Data structure - Sat4j provides cardinality and binary data structures as well as H/T and WL data structures [16].

- Order selection - The decision heuristics available to choose from are natural order and VSIDS as well as some extensions of it [20].
- Phase selection - There are several possible strategies to select the phase (or polarity) of the variable to branch on first. It can be fixed (negative or positive first, user-defined), random, or based on previous assignments (phase saving [21]).
- Random walk - To escape from the bad choices of the heuristics, a common practice is to pick a variable at random, which is referred to as making a random walk. Sat4j allows to add random walks with a given probability to order heuristics.
- Learning Strategy - The available learning strategies in Sat4j include fixed and percent length learning, active learning, and Minisat learning. It is also possible to not use any learning strategy.
- Learnt constraints deletion strategy - The measures of the importance of the learnt clause that can be used are activity or Literals Blocks Distance (LBD), as defined in Glucose [18]. Sat4j allows the user to also clean up the database periodically or on demand.
- Conflict-clause minimization - Minisat 1.13 introduced two conflict minimization procedures to reduce the clauses derived by conflict analysis. Sat4j implements both simple and expensive simplifications. The features can also be deactivated.
- Restart strategy - Sat4j provides only static restart strategies (the one inherited from Minisat, the one used in Picosat [23], and a Luby-style one [22]). However, it also allows the end user to decide when to restart.

Advanced SAT solver users typically utilize various metrics to assess the most effective solver or settings for a specific instance. To address this Sat4j displays in real-time

several metrics from the solver that are important to understand the behavior of the solver including the decision index (allows to check if the decisions are limited to a group of variables or if they are spread across all variables), activity value (useful to check that the heuristics work as expected), size of learnt clauses (used to check whether minimization techniques are effective or not), evaluation of learnt clauses, speed (number of propagations per second), decision level (the depth in the search tree), and trail level (the number of variables assigned when reaching a conflict) [27]. The idea is to enable the user to adapt the configurations of the solver for their needs in light of those metrics [27].

2.3 Fuzzing

The original fuzzing approach aimed to detect security bugs by testing programs with randomly generated inputs. Fuzz testing is a form of blackbox random testing that randomly mutates well-formed inputs and tests the program on the resulting data. To randomly generate the well-formed inputs *grammar* is used where a model describes the set of syntactically valid inputs [4, 29]. An alternative approach, whitebox fuzz testing, is built upon recent advances in dynamic symbolic execution [4].

Klees et al. [30] showed that performance can vary substantially depending on the generated seeds. Fuzzers used to test SAT solvers must generate interesting problems that are fast to solve, this is especially important if heavy testing will be done on clusters where running more problems is more important than running fewer problems for longer.

2.3.1 Random Generators in Java

Randomness is a key feature of fuzzing and there are 2 random generation classes provided by Java used in this project: `SecureRandom` and `Random` [31]. `SecureRandom`

provides a cryptographically strong random number generator (RNG). SecureRandom produces non-deterministic output, therefore any seed passed to a SecureRandom object must be unpredictable, and all SecureRandom output sequences are cryptographically strong. An instance of a Random class is used to generate a stream of pseudorandom numbers. If two instances of Random are created with the same seed, and the same sequence of API calls is made for each, they will generate and return identical number sequences, thus they are deterministic.

2.4 Delta Debugging

The first step in tackling any bug is simplification, eliminating all irrelevant details for producing the failure. A simplified bug report facilitates debugging and subsumes several other reports that differ in irrelevant details [32]. It is possible to automate this simplification and automatically isolate the differences that cause failures. One of the algorithms available to do this is Delta Debugging [32].

Presented by Zeller and Hildebrandt [32], the basis of Delta Debugging is binary search. If c contains only one change then c is minimal, otherwise partition c into two subsets with similar size and test each of them. Let n be the number of subsets $\Delta_1; \dots \Delta_n$. Testing each Δ_i and its complement $\nabla_i = c_x - \Delta_i$ there are four possible outcomes as presented in Equation (2):

- Reduce to subset - If testing any Δ_i fails then continue reducing Δ_i with $n = 2$ subsets. If it can identify a smaller part of the failure-inducing test case, then this rule helps narrow the test case significantly and efficiently.
- Reduce to complement. If testing any $\nabla_i = c_x - \Delta_i$ fails then continue reducing ∇_i with $n - 1$ subsets because the granularity stays the same.
- Increase granularity - Try with $2n$ subsets. This results in at most twice as many tests, but increases chances for failure.

- Done - The process is repeated until granularity can no longer be increased. In this case, it has already tried removing every single change individually without further failures. The resulting change set is minimal.

$$dd(c_x, n) := \begin{cases} dd(\Delta_i, 2) & \text{if } \exists i \in \{1, \dots, n\} \cdot test(\Delta_i) = X \\ dd(\nabla_i, max(n-1, 2)) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(\nabla_i) = X \\ dd(c_x, min(|c_x|, 2n)) & \text{else if } n < |c_x| \\ c_x & \text{otherwise ("done")} \end{cases} \quad (2)$$

Delta Debugging algorithm [32]

Brummayer et al. [6] showed that fuzz testing and delta debugging combined are quite effective in testing and debugging large SAT solver implementations with a high degree of automation. The basic workflow is shown in Figure 5. It consists of the test case generator for creating random formulas according to a grammar provided by a data model (i.e. the fuzzer) and the delta debugger used in case of failures to reduce the size of the formula such that the failure still persists [29].

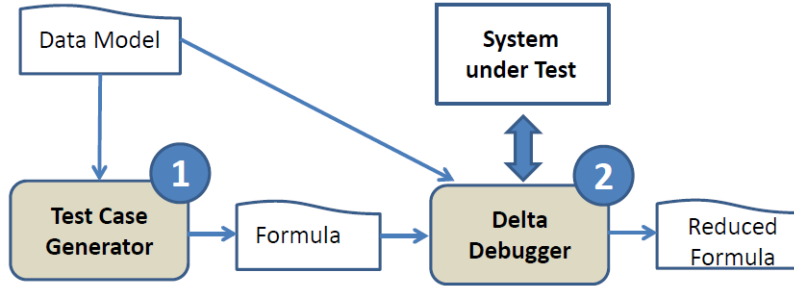


Figure 5: Fuzzing and Delta Debugging Workflow [29].

3 Related Work

This chapter presents API fuzzing, how it differs from input fuzzing, and acknowledges some existing tools that use API fuzzing for SAT and SMT solvers including how they work and perform. This chapter also shows how incremental proof-checking is done using the iCNF and IDRUP file formats introduced in Chapter 2.

3.1 API Fuzz Testing

SAT solvers usually provide an application programming interface (API) as the direct connection between the application and its solver back-end. This API often introduces additional functionality not supported by the input language so it may not be possible to test the full feature set a solver provides simply by generating randomized valid input sequences. In model-based testing, test cases are derived from an abstract model rather than implemented directly in the code [33]. This approach has several advantages: a high-level model is easier to develop and understand and partially specified behaviors give rise to many possible combinations, from which many concrete test cases can be derived [33].

Artho et al. [29] propose the use of a model-based testing approach for verification back-ends like SAT solvers. They suggest fuzzing not only the input data but also generating sequences of API calls to cover more features. This sequence of valid API calls is described by a state machine. It is also possible to test different solver features by fuzzing the different configurations specified in the state machine.

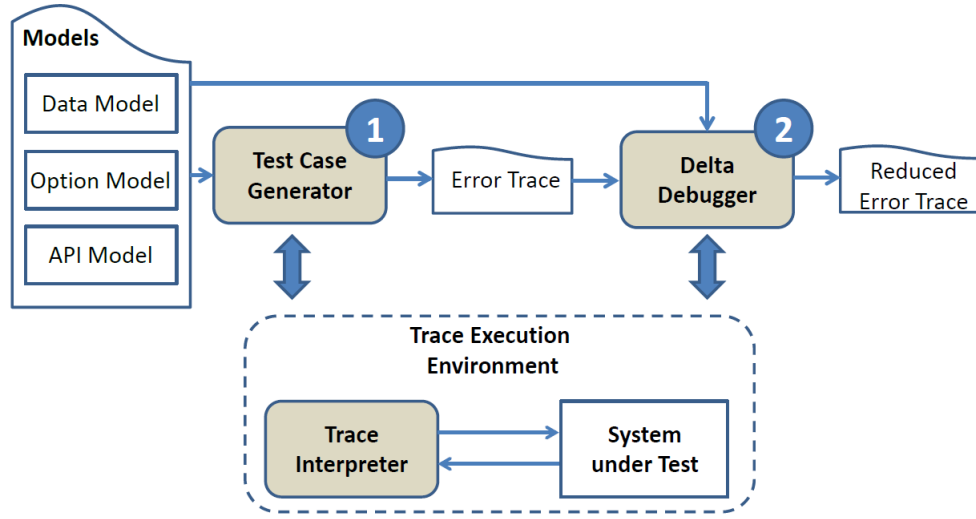


Figure 6: Fuzzing and Delta Debugging Extended Workflow [29].

Figure 6 shows the extended fuzzing and delta debugging workflow, now with 3 main components: test case generator, delta debugger, and trace interpreter.

The test case generator accepts as input three different models: the data model provides the grammar for generating valid formulas, the option model describes valid feature configurations and combinations, and the API model describes valid traces of API calls that can be made to the solver.

The delta debugger reduces the trace that contains the input formula and the API calls made to the solver so that the failure persists. It has to obey the API model description to maintain a valid trace. The delta debugger communicates with the system under test to receive the results of the reduction process.

The trace interpreter has to be developed for each SUT (system under test) individually and can call the functions of the SUT directly. The output of the SUT is translated into a format that is processed by the testing framework. The trace interpreter replays the trace reduced by the delta debugger which can then undergo manual debugging to identify and eliminate defects.

The results presented by Artho et al. [29] show that this approach produces higher code coverage and throughput (number of test cases executed per second) and that it can detect a larger number of code mutations than file-based input fuzzing. They conclude that the model-based testing approach is substantially more effective in finding defects than previously used techniques. Most of the existing implementations for this framework are done in C or C++ so the motivation behind this thesis is to implement this framework for the only competitive SAT solver in Java, Sat4j.

The framework has also been adapted for SMT solvers (Satisfiability Modulo Theory, generalized SAT solvers, capable of handling various data types). BtorMBT proved to be quite effective where triggered errors include failed internal checks, assertion failures, segmentation faults, and any other kind of abort [34]. Murxla is another tool developed for testing SMT solvers that quickly and effectively found issues in multiple state-of-the-art solvers, even for logics that have been the subject of other fuzzing campaigns over the years [35]. BanditFuzz, a multi-agent reinforcement learning (RL) guided performance fuzzer for state-of-the-art SMT solvers was able to expose surprising performance deficiencies in these tools [36].

3.2 Incremental Proof-Checking

Section 2.1.5 explains what proofs are, this section presents the process of incremental proof-checking. Proofs can be checked either forward or backward. Forward checking verifies each clause addition step in the same order as they were done by the solver and can be done online during solving [10]. Backward checking verifies clause addition steps in reverse order, starting from the empty clause, therefore, it can start only once solving is finished, but it allows the checker to check only those clauses that are used in the refutation [10].

Extending an already built and solved SAT formula with further constraints, instead of constructing and solving new formulas in each step, allows the reuse of the same SAT solver instance. However, there are no widely used proof-checking techniques for incremental usage even though incremental solving adds another level of complexity and thus can be considered even more error-prone [10, 37]. Fazekas et al. [10] propose both a format and a way of checking incremental proofs. Their implementation was done for CADICAL, a SAT solver implemented in C++. This thesis aims to integrate this proof-checker in the API fuzzing framework implemented in Sat4j.

The state of proof-checking can be captured on an abstract level by two multi-sets of clauses: the active clauses F_A and passive clauses F_P , both initially assumed to be empty. Given an IDRUP file, Equation (3) describes how each line L_i updates the current state of the proof-checker. Input clause addition steps by the user i add a clause to the active set and need to be validated that it is indeed the next added clause in the iCNF file. Lemma addition steps l need to be checked to have RUP w.r.t. the current active formula (F_A^i). Clause deletion d and weakening steps w are only allowed to remove a clause if the clause is currently present in F_A^i . In the case of weakening the clause is made passive while for deletion it is just dropped. Similarly, a restore step r can be applied only on a clause if it is currently present in the passive formula (F_P^i). Both F_A^i and F_P^i are multi-sets, i.e. in theory the same clause can be added, removed, and restored multiple times and a proof-checker must be able to handle multiplicity correctly.

Status lines s do not modify the state of the proof-checker but serve as synchronization points between the proof and the interaction file. In case the answer is satisfiable the following m line defines a satisfying assignment. This assignment is checked whether it satisfies not just every input clause provided until that point in the iCNF file, but also every assumption of the current query. Unsatisfiable answers are justified by the following u line, which is empty if there were no assumptions or defines a subset of the assumptions of the current query that makes the problem inconsistent.

$$(F_A^{i+1}, F_P^{i+1}) := \begin{cases} (F_A^i \cup \{L_i\}, F_P^i) & \text{if } t(L_i) = "i" \\ (F_A^i \cup \{L_i\}, F_P^i) & \text{if } t(L_i) = "l" \\ (F_A^i \setminus \{L_i\}, F_P^i) & \text{if } t(L_i) = "d" \\ (F_A^i \setminus \{L_i\}, F_P^i \cup \{L_i\}) & \text{if } t(L_i) = "w" \\ (F_A^i \cup \{L_i\}, F_P^i \setminus \{L_i\}) & \text{if } t(L_i) = "r" \\ (F_A^i, F_P^i) & \text{otherwise} \end{cases} \quad (3)$$

Incremental Proof Checking [10]

To demonstrate that their approach is practically viable, Fazekas et al. [10] implemented the IDRUP proof-checker IDRUP-CHECK [11]. The proof-checker reads and checks both the iCNF and the IDRUP file in parallel according to the state machine shown in Figure 7. After checking headers in both files the input clauses are read from the iCNF interactions file and matched against input lines (in the same order) in the IDRUP proof file. Proof-checking is only successful if all lines match in both files, the implicit checks described above succeed and all queries observed in the iCNF file (the larger and golden q) have matching justifications (the larger and green m or u lines) in the IDRUP file.

To evaluate the proof of concept implementation Fazekas et al. [10] extended CAMICAL, a bounded hardware model checker, to produce IDRUP proofs and iCNF files to then check them with IDRUP-CHECK [11]. It turned out that proof production has a relatively small overhead compared to plain solving. It takes around a factor of two for proof-checking compared to plain solving, which is considered reasonable as it is in the order expected for DRUP proofs (in the SAT competition solving time limit is 5,000 seconds while proof-checking is 40,000 seconds). Their experiments show that the format is practical and has the potential to support efficient certification and verification of incremental use cases of modern SAT solvers.

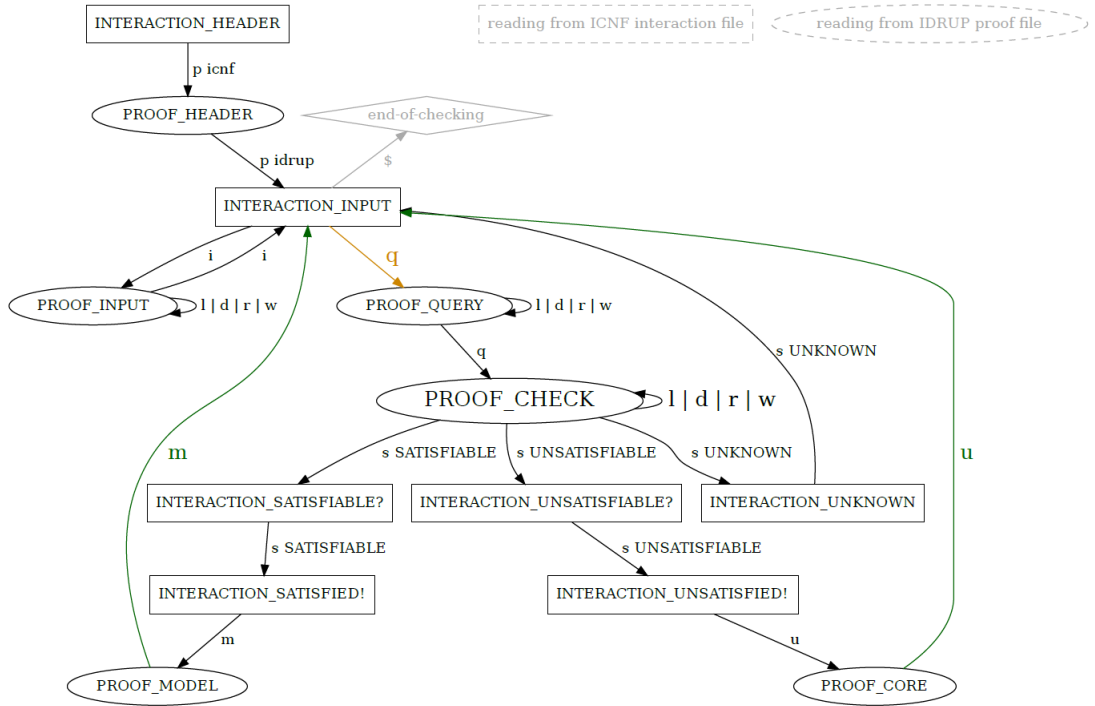


Figure 7: The state-machine based implementation of IDRUP-CHECK interleaves parsing and checking lines from the iCNF interactions file (rectangular states) and from the IDRUP proof file (oval states) [10].

4 Workflow Implementation in Sat4j

This chapter presents the details of implementing model-based API fuzzing in Sat4j. The test case generator includes clause generation, incremental solving, assumptions, and different configurations of solver features. It is essential to verify that the solver’s results are correct, which is why, in collaboration with Prof. Dr. Le Berre, we added incremental DRUP proofs as well as incremental CNF generation to support proof-checking via the external proof-checker IDRUP-CHECK [11] provided by Prof. Dr. Biere. The tool also provides a trace interpreter and a delta debugger for the complete workflow presented in Figure 6.

4.1 Test Case Generator

Sat4j already relied on unit tests, assertions, and regression testing but not on fuzzing [12]. Since Sat4j provides a rich API interface, it was more beneficial to implement API fuzzing instead of input fuzzing. For this, I followed the framework presented in Section 3.1. The first step was implementing the test case generator, i.e. the fuzzer, which creates valid sequences of API calls, also known as traces, based on the provided data, options, and API models.

Since randomness is a crucial part of fuzzing I make use of both the `SecureRandom` and `Random` libraries from Java as mentioned in Section 2.3.1. When the fuzzer is launched I create an instance of `SecureRandom` and use it to create a “master” that will always be unique since `SecureRandom` is non-deterministic. I use the “master”

seed to create an instance of Random known as the “master random generator” which generates all the “slave” seeds. Each “slave” seed represents a unique API sequence. I create a “slave random generator” from the “slave” seed which is responsible for all the decisions including the number of variables, clauses, increments, assumptions, the solver used, and its configurations. Whenever an error occurs a “trace” file is created with the HEX value of the “slave” seed and logs all the API calls made up until the error was thrown.

It is possible to provide the “master” seed to the fuzzer to make it deterministic or replay a particular run of “slave” seeds. The default number of seeds generated per execution is set to 100 but it is possible to change that by providing the fuzzer the new number via the command line. It is also possible to tell the fuzzer to print information during execution and skip proof-checking if needed.

4.1.1 Option Fuzzing

The first API call made on every trace is `init` which initializes the default solver `Glucose21`, considered the best-performing solver (Section 2.2). After the solver is initialized the configuration features, i.e. options, are fuzzed. The option model I provided allows the fuzzer to use all options, to use some options randomly, or to not use any options at all. Based on that outcome the fuzzer might use one of the pre-defined solvers or randomly select feature configurations.

The `SolverFactory` [12] in `Sat4j` provides 33 different pre-defined solvers that include a variety of different configurations. Some of them include `Backjumping` with `VSIDS` heuristics set to true, clause generator for backjumping set to true and learning set to false; `RandomSolver` with `VSIDS` heuristics set to true, random walk set to true, learning set to false, and restarts set to false; `Light` with learning set to true, and so on [12]. Several multi-core solvers initiate 2 to 8 solvers and run them in parallel in multiple cores.

Algorithm 2 Option Fuzzing Algorithm

```
solver = defaultSolver()
if randomGenerator.nextBoolean() then                                ▷ use no options
    useAll = randomGenerator.nextBoolean()                            ▷ use all options
    if useAll || randomGenerator.nextBoolean() then
        if randomGenerator.nextBoolean() then
            solverName = SOLVERS.get(randomGenerator.nextInt())
            solver = createsSolverByName(solverName)
        else
            solver = configureSolverFeatures(randomGenerator)
        end if
    end if
    if useAll || randomGenerator.nextBoolean() then
        probability = randomGenerator.nextDouble()
        solver.setRandomWalk(probability)
    end if
    if useAll || randomGenerator.nextBoolean() then
        solver.setDBSimplificationAllowed(randomGenerator.nextBoolean())
    end if
end if
return solver
```

When creating a SAT solver there are several configurable features provided by Sat4j which include data structure factory, order selection strategy, phase selection strategy, learning strategy, restart strategy, simplification, search parameters, learnt constraints evaluation, random walk, and DB simplification.

The data structure factory option defines a concrete implementation of clauses, cardinality constraints, and pseudo boolean constraints. While working on the project I added a constraint to the option model for cardinality data structures so that a unit clause literal is not used in its negated form. This restriction was noticed when the solver would throw a “Contradiction With Implied Literal Exception” when trying to add a clause to the formula that contained the negated literal of an earlier unit clause as can be seen in Trace 1 in Figure 8.

Another constraint I added to the option model was for `MixedDataStructureDanielWLConciseBinary` [12] where it was necessary to provide the maximum variable of the formula to the solver before adding clauses. Trace 2 in Figure 8 is an example of a reduced trace that helped identify this restriction. When adding clause $-235\ 3$ the solver would throw an “Array Index Out Of Bounds Exception” since the maximum variable in the solver at that moment was 234.

Trace 1
1 init
2 Data Structure Factory : CardinalityDataStructureYanMax
294 addClause 1
772 addClause -1
Trace 2
1 init
2 Data Structure Factory : MixedDataStructureDanielWLConciseBinary
327 addClause 234
339 addClause 2 1
687 addClause -235 3

Figure 8: The reduced traces that helped identify the option model constraints for cardinality data structures and `MixedDataStructureDanielWLConciseBinary` [12] data structure.

Simplification decides if the solver is allowed to simplify the reason, i.e. learnt clause, and if so then how much time is allocated for it. Sat4j provides four options: no simplification is used, simple simplification, expensive simplification, and expensive simplification of watched literals only. A restriction was detected when the fuzzer combined `MixedDataStructureDanielWLConciseBinary` [12] data structure and `expensiveSimplificationWLOnly` [12] simplification. The solver would throw an “Unsupported Operation Exception” when solving the trace in Figure 9. In `expensiveSimplificationWLOnly` the constraints are supposed to be only based on watched literals so the analysis would abort for the compact binary clauses. I added a constraint to the option model to replace the simplification type with `expensiveSimplification` [12] when this combination is encountered.

Trace
1 init
2 Data Structure Factory : <code>MixedDataStructureDanielWLConciseBinary</code>
7 Simplification Type : <code>expensiveSimplificationWLOnly</code>
30 addClause -5 1
136 addClause -3 2
144 addClause 3 4
148 addClause 5 -3 -2
208 solve

Figure 9: The reduced trace that identified the option model constraint for the simplification and data structure factory combinations.

In addition to adding support for the different features of the solver in the fuzzer, I also included the parameters used in these configurations in the option model. This means that the fuzzer will not simply use the default values encoded in Sat4j but randomly select a valid value in a specified range. This could lead to slow solvers in certain cases so it proved necessary to add a timeout. Each solver is assigned a timeout of 2 minutes when initiated.

Two order selection strategies include parameters that can be fuzzed. The VarOrderHeap [12] strategy uses VSIDS-like heuristics with a variable decay parameter which I fuzz in a range from 0 to 1 (excluded). The PureOrder [12] strategy is an extension of VarOrderHeap [12] that tries to branch on a single phase watched unassigned variable every x decisions where x is the period parameter fuzzed in a range from 0 to 100. Sat4j incorporates random walk with the chosen order selection strategy, meaning that sometimes when deciding the variable to branch it can be randomly selected. How often this randomness is utilized is defined by setting its probability which I fuzz from 0 to 100% (excluded).

For learning strategies, there are three configurations with changeable parameters. The ActiveLearning [12] strategy limits learning only clauses containing x percentage of active literals where x is fuzzed from 0 to 100% (excluded). The FixedLengthLearning [12] strategy limits learning clauses of size smaller or equal to x maximum length where x is fuzzed from 0 to the number of variables in the formula. PercentLengthLearning [12] strategy limits learning clauses of size smaller or equal to x percent of the number of variables where I fuzz x from 0 to 100%.

There are two restart strategies with configurable parameters. FixedPeriodRestarts [12] strategy defines constant restarts every x conflicts where x is fuzzed from 0 to 10,000. LubyRestarts [12] strategy defines Luby style restarts with factor x where x is fuzzed from 0 to 1,000.

Sat4j makes it possible to define in the solver some of the parameters it uses during searching in the SearchParams [12] class. These parameters are also included in the option model provided to the fuzzer. Variable decay (varDecay) and clause decay (claDecay) are fuzzed in the range from 0.9 to 1.0 (excluded). The initial conflict bound for the first restart (initConflictBound) is fuzzed in the range from 0 to 1,000 and the conflict bound increase factor (conflictBoundIncFactor) is fuzzed in the range from 0 to 3.0 (excluded).

4.1.2 Clause Generation

As mentioned in Section 2.1 SAT solvers accept input in the CNF format which is a conjunction of clauses where each clause is a disjunction of literals and each literal is either a plain or negated variable. The data model provided to the fuzzer defines the proper grammar for generating valid clauses that are then passed to the solver.

Before generating the clauses I had the fuzzer first decide the number of variables (MAXVAR) and the number of clauses of the formula. The number of variables is randomly selected in the range between 20 to 200. The number of clauses is based on a coefficient optimized to generate 50% satisfiable and 50% unsatisfiable instances. A higher coefficient means more clauses, which means more UNSAT problems. After testing, it was seen that a coefficient of 2.6 produced the best results when taking into account the number of increments as well. The number of clauses is generated by multiplying this coefficient with the number of variables.

The data model is designed to support a uniform clause length of three as well as varying clause lengths based on a power law, however, even in varying length configurations, the majority of clauses will still be of length three. The fuzzer will generate 1% of clauses as unit clauses, i.e. clauses of length one, 10% of clauses as binary clauses, i.e. clauses of length two, and about $\sim 17\%$ of clauses with a length higher than three. Whenever the length of the clause is increased there is a $\sim 17\%$ chance of it increasing again.

After deciding the length of the clause the literals are generated one by one in the range -MAXVAR to MAXVAR, excluding 0. I added a constraint in the data model so the fuzzer is not allowed to use the same literal twice in a clause. This was a restriction from Sat4j, as the clause is simplified by the solver automatically, but by the IDRUP-CHECK [11] proof-checker which was having difficulty in following the reasoning steps and was throwing a “Lemma Implication Check Failed Exception”.

In case the cardinality constraint is needed the fuzzer makes sure not to use a literal that has been added as a unit clause in a negated form. All this can be observed in Algorithm 3.

Algorithm 3 Clause Generation Algorithm

```

clauseLength = 3
uniform = randomGenerator.nextDouble()
if !uniform then
    percentage = randomGenerator.nextDouble()
    if percentage < 0.01 then
        clauseLength = 1
    else if percentage < 0.1 then
        clauseLength = 2
    else
        while percentage < 1/6 do
            clauseLength += 1
            percentage = randomGenerator.nextDouble()
        end while
    end if
end if
clause ← int[clauseLength]
for j = 0 to clauseLength do
    literal = randomGenerator.nextInt(2 · MAXVAR) − MAXVAR
    while literal == 0 || isPresent(clause, literal) || cardinalityCheck do
        literal = randomGenerator.nextInt(2 · MAXVAR) − MAXVAR
    end while
    clause[j] = literal
end for

```

4.1.3 Assumptions

Assumptions are literals presumed to be true. If one or more of the assumed literals fails to hold during the search then the solver returns UNSAT and provides the list of failing assumed literals, known as the UNSAT core.

The API interface of Sat4j [12] allows passing a list of literals to the solver when solving which are treated as assumptions. The fuzzer decides if assumptions will be used and decides on the number of literals that will be assumed. Like clause generation, the assumptions start at 1/10th of the number of variables and then there is a $\sim 17\%$ chance of it increasing by 1/10th of the number of variables each time. The assumptions are generated from -MAXVAR to MAXVAR excluding 0. The data model constraints also apply to assumptions so the fuzzer ensures that the same literal is not assumed twice and keeps track of all literals in the formula to avoid assuming a literal that is not present.

4.1.4 Enumeration Fuzzing

The fuzzer was extended to support fuzzing for the internal and external enumerators provided by Sat4j [12]. This was motivated by an existing bug where users had reported that the internal model enumerator provided incorrect results on certain input files [38]. When testing enumeration results the fuzzer will create two instances of the solver, one will make use of the internal enumerator and the other will connect to the external enumerator. The solvers are configured with the same features and are provided the same clauses.

The data model will generate a simpler formula when enumerating with up to 20 new variables, compared to the normal 20 to 200 variables. I added this condition because when enumerating the fuzzer has to solve the formula repeatedly and a complicated formula had a higher probability of leading to timeout and not being useful. Since enumeration formulas have fewer variables, the coefficient is increased to 5.

It is important to note that the internal enumerator returns the number of models based solely on the used variables in the input formula while the external enumerator takes into consideration the unused variables as well. I encountered this when analyzing one of the traces with only one input clause, $6 - 2$. The external enumerator would not count 3 models ($6 - 2$; $6 \ 2$; $-6 - 2$) but 48 due to the four unused variables that can be combined in 16 different ways, so the actual result of 3 gets multiplied by 16 to make 48. This created a lot of enumeration traces that were not actually faulty. I updated the fuzzer to keep track of used variables in the formula and compare it with the number of variables in the solver, retrieved via the provided API. This information is used to determine the number of unused variables, calculate their combinations, and divide the result returned by the external enumerator with it. The fuzzer now throws an error if the result of the internal enumerator differs from the updated result of the external enumerator.

Parallel solvers are excluded if the fuzzer is testing enumerators due to race conditions that were observed to affect the results. MinOneSolver [12] is also excluded as it computes the models with a minimum number of satisfied literals and not one solution at a time like.

4.1.5 Incremental Solving

Sat4j supports increments when solving but not when enumerating. This is because when enumerating the fuzzer finds all the models and increments want only one solution. With normal solving the fuzzer will add 20 to 200 new variables to the formula with each increment. The number of clauses added in each increment is based on the number of new variables and the coefficient. If it is needed, the new maximum number of variables is passed to the solver. The clauses are generated as explained in Section 4.1.2. If the fuzzer is solving with assumptions then the assumptions are generated as explained in Section 4.1.3 and the solver starts searching for a solution. All this can be seen in Algorithm 4.

Algorithm 4 Incremental Solving Algorithm

```
ENUMERATING = randomGenerator.nextInt(5) == 0
ASSUMPTIONS = randomGenerator.nextBoolean()
if ENUMERATING then
    MAXVAR = randomGenerator.nextInt(20) + 1
    coefficient = 5
else
    MAXVAR = randomGenerator.nextInt(181) + 20
    coefficient = 2.6
end if
numberOfClauses = coefficient · MAXVAR
solver = initializeSolver()
solverForExternalCounter = initializeSolver()
totalIncrements = randomGenerator.nextInt(5) + 1
for increments = 1 to totalIncrements do
    if increments != 1 then
        oldMAXVAR = MAXVAR
        MAXVAR = randomGenerator.nextInt(181) + 20 + oldMAXVAR
        numberOfClauses = coefficient · (MAXVAR - oldMAXVAR)
    end if
    if passMAXVAR then
        solver.newVar(MAXVAR)
        if ENUMERATING then
            solverForExternalCounter.newVar(MAXVAR)
        end if
    end if
    addClauses()
    if ENUMERATING then
        internalCount = Helper.countSolutionsInt(solver)
        externalCount = Helper.countSolutionsExt(solverForExternalCounter)
        if internalCount != externalCount then
            throw error
        end if
        break
    else
        if ASSUMPTIONS then
            assumptions = generateAssumptions()
            isSAT = solver.isSatisfiable(assumptions)
        else
            isSAT = solver.isSatisfiable()
        end if
    end if
end for
```

4.1.6 Integrating IDRUP-CHECK

Sat4j supports DRUP proof format but for the fuzzer to support incremental proof-checking Sat4j needed to support incremental proof generation. It was decided with Prof. Dr. Le Berre that a new listener would be created. When attached to the solver the listener will log all the interactions between the user and the solver as well as the reasoning decisions made when solving in the IDRUP format presented in Section 2.1.5. Prof. Dr. Le Berre implemented the needed changes in Sat4j [12].

The fuzzer takes care of generating the iCNF file as presented in Section 2.1. I implemented this by logging all the clauses added to the solver, the assumptions, and the result returned from the solver together with the model or UNSAT core it provides respectively, and writing them in a file after the final increment.

Once incremental solving has finished the IDRUP and iCNF files are passed to the incremental proof-checker IDRUP-CHECK [11] which follows the logic presented in Section 3.2. If the checker exits with code 0 then the check is successful and the iCNF and IDRUP files are deleted to not overload the memory of the machine. If an error is thrown from the checker the files are kept to recreate and analyze the issue.

There are a few occasions when proof-checking is skipped. If an error occurs when initializing the solver, adding clauses, or solving then the iCNF and IDRUP files are not complete so proof-checking is not possible. When the fuzzer is enumerating all the models would be logged in the proof which was not compatible with IDRUP-CHECK [11]. If MinOneSolver [12] is being used it computes the models with a minimum number of satisfied literals and much like when enumerating the generated proof is incompatible with the proof-checker. As mentioned in Section 4.1.1 parallel solvers initiate 2 to 8 different solvers that run in multiple cores but share the same listener, meaning they all add their interactions to the same IDRUP file. This together with race conditions means that the proof can not be properly generated and the proof-checker cannot properly verify it.

Figure 10 shows part of a proof generated by the Parallel [12] solver that initiates 8. The first issue is the multiple lines of queries logged in the proof by all the solvers. When running IDRUP-CHECK [11] with this proof and its respective iCNF file it throws an “Unexpected q Line Exception”. The second issue is the status (s) and model (m) lines which are not recorded correctly due to race conditions from the different threads. Extending the proof generation to parallel solvers proved to be outside the scope of this thesis.

Partial IDRUP proof
q 0
q 0
q 0
q 0
q 0
q 0
q 0
q 0
s SATISFIABLE
m s SATISFIABLE
m -1 -2 -3 4 5 6 7 -8 -9 10 -11 12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 0
s SATISFIABLE
s SATISFIABLE
m -1 m -1 -2 -3 4 5 6 7 -8 -9 10 -11 12 13 -14 -15 -16 -17 -18s SATISFIABLE
m -1 -2 -3 4 5 6 7 -8 -9 10 -11 12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 0
s SATISFIABLE
m -1 -2 -3 4 5 6 7 -8 -9 10 -11 12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 0
s SATISFIABLE
m -1 -2 -3 4 5 6 7 -8 -9 10 -11 12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 0
-19 -20 -21 -22 -23 -24 -25 -26 0
-2 -3 4 5 6 7 -8 -9 10 -11 12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 0
s SATISFIABLE
m -1 -2 -3 4 5 6 7 -8 -9 10 -11 12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 0

Figure 10: Partial IDRUP Proof generated from Parallel [12] solver.

4.1.7 Statistics

As I mentioned in Section 2.2 Sat4j provides useful information on the status of the search. I use this information to print statistics for all the seeds that successfully finished executing and proof-checking. Below is an example of said statistics:

```
c Statistics for 100 iterations :  
c Error Instances : 0  
c Timeout Instances : 0  
c SAT Instances : 33  
c UNSAT Instances : 44  
c ENUM Instances : 23  
  
c Average Propagations : 6002  
c Average Decisions : 300  
c Average Starts : 2  
c Average Reduced Literals : 256  
c Average Learned Clauses : 110  
c Average Updates of LBD : 9  
c Average Conflicts : 111  
c Average Solver Run Time per Increment : 0 milli sec  
c Average Proof-Checking Time : 55 milli sec
```

Statistics are not retrieved from the solver when the fuzzer is testing the internal and external enumerators. The provided statistics include information on the number of propagations, decisions, and restarts as well as the number of conflicts, learned clauses, and LBD updates. The run time for the trace is calculated for each increment and then divided by the number of increments.

4.2 Trace Interpreter

The second step in supporting API fuzzing is implementing the trace interpreter, i.e. the `TraceRunner`, as presented in Figure 6. This tool provides replay functionality, enabling users to reproduce errors and analyze error traces. It is also a simple way to generate a single trace.

If a seed is passed as the argument from the command line, the `TraceRunner` [39] will call the `TraceFactory` [39] and configure it to treat the seed as a “slave” seed and go through the process explained in Section 4.1. The execution will end after running the seed and if an error is thrown a trace will be created with the HEX representation of the seed.

If a trace file is passed as the argument from the command line then the file is parsed and each line representing an API call is executed in sequence. Algorithm 5 shows how `TraceRunner` [39] iterates through all the parsed API calls and if needed executes the proof-checker in the end. When running the file, it is unknown if the trace is an enumeration error, so two solvers are created, configured, and connected with the IDRUP proof listener even though it may not be necessary.

4.3 Delta Debugger

The third step in completing the API fuzzer is implementing the delta debugger. As explained in Section 2.4, delta debugging simplifies failure-inducing input by removing redundant parts to produce a minimal file. I have extended the algorithm to not only remove unnecessary lines as in Equation (2) but also rename literals and remove excessive literals from clauses and queries. `TraceRunner` [39] presented in Section 4.2 is utilized to run the complete trace and sub-sections of it during the elimination process.

Algorithm 5 TraceRunner Algorithm

```
for i = 0 to apiCalls.size() do
  if apiCalls.get(i) == null then
    continue
  else if apiCalls.get(i).contains("using solver") then
    solverName = getSolverName(apiCalls.get(i))
    solver = initializeSolver(solverName)
    solverForExternalCounter = initializeSolver(solverName)
  else if apiCalls.get(i).contains("feature") then
    featureConfiguration = getFeatureConfiguration(apiCalls.get(i))
    solver.setFeature(featureConfiguration)
    solverForExternalCounter.setFeature(featureConfiguration)
  else if apiCalls.get(i).contains("newVar") then
    MAXVAR = getMAXVAR(apiCalls.get(i))
    solver.newVar(MAXVAR)
    solverForExternalCounter.newVar(MAXVAR)
  else if apiCalls.get(i).contains("addClause") then
    clause = getClause(apiCalls.get(i))
    solver.addClause(clause)
    solverForExternalCounter.addClause(clause)
  else if apiCalls.get(i).contains("assuming") then
    assumptions = getClause(apiCalls.get(i))
    solver.isSatisfiable(assumptions)
  else if apiCalls.get(i).contains("solve") then
    solver.isSatisfiable()
  else if apiCalls.get(i).contains("enumerating") then
    skipProofCheck = true
    internalCount = Helper.countSolutionsInt(solver)
    externalCount = Helper.countSolutionsExt(solverForExternalCounter)
    if internalCount != externalCount then
      throw error
    end if
  end if
end for
if !skipProofCheck then
  Helper.createICNFfile()
  process = Runtime.getRuntime().exec(idrup-check)
  exit = process.waitFor()
  if exit != 0 then
    throw error
  end if
end if
```

DeltaDebugger [39] first tries to remove complete API calls from the trace. Once no more lines can be removed it iterates through all the API calls that add clauses or assume literals and tries to remove each of those literals one by one. Once no more literals can be removed it tries to rename the remaining literals to smaller numbers. If a formula has only one clause with literals 23 87 then they are renamed to 1 and 2 respectively if the error persists. If any of these processes modify the trace they are all repeated until a final minimal version of the trace is reached.

Figure 11 shows a snippet of a trace containing 721 API calls and its reduced version with only 11 API calls from delta debugging. The *init* line is never removed since it creates the solver. The API call with index 7 two literals have been removed from the clause. The original trace has up to 240 literals while the reduced file has only 10 and they have been renamed.

Trace	Reduced Trace
1 init	1 init
2 addClause 14 16 -18	7 addClause 1
3 addClause -21 -4 13	11 addClause -8 -6
4 addClause -6 -25	35 addClause 6 -5 6
5 addClause 10 -27 -26	42 addClause 8 7
6 addClause -7 -21 13	60 addClause -7 2
7 addClause -25 1 22	61 addClause -7 -4
.....	63 addClause -3 5
627 assuming 8 -116 ... -117 -128	72 addClause -2 4 9
.....	74 addClause 9 3
720 addClause 24 -76 -238	721 assuming -10
721 assuming -37 203 ... -20 -33	

Figure 11: A trace generated by the fuzzer (721 lines and 2 incremental calls) and its reduced form (11 calls and 1 call) from running DeltaDebugger [39].

Algorithm 6 Extended Delta Debugging Algorithm

```
content = Files.readAllLines(Paths.get(fileName))
errorType = TraceRunner.runTrace(content)
if errorType == null then
    return
end if
tryAgain = true
while tryAgain do                                     ▷ All 3 methods set tryAgain
                                                         to true if the trace is updated
    tryAgain = false
    removeLines()
    removeLiterals()
    renameLiterals()
end while
output = TraceRunner.runTrace(content)
if output != null && output.equals(errorType) then
    createReducedFile()
else
    return error
end if
```

5 Results

This chapter presents the results of adding model-based API fuzzing in Sat4j. The first section contains the coverage reports achieved by the fuzzer. The following sections detail the bugs found from fuzzing and proof-checking, how the related trace and proof files were used to find the root cause, and how the issues were fixed.

5.1 Code Coverage

One metric for determining the effectiveness of the API fuzzer is code coverage. Since the fuzzer does not test all the capabilities offered by Sat4j it also does not provide coverage for all classes in the library. It focuses on the SolverFactory class that contains all the pre-defined solvers, the folders containing the different options, and the classes used for internal and external enumerating [12].

The coverage report is generated with JaCoCo, a free Java code coverage library [40]. To generate only the coverage of the API fuzzer I created a new unit test (TestAPIGenerator [39]) that initiates the fuzzer and runs it until it generates 200 traces. Due to randomness, it is unlikely to get full coverage of all the configurations without running the fuzzer for a long time. It is possible to access the detailed report at org.sat4j.core/target/site/jacoco folder in the project repository [39]. Figure 12 shows the coverage report for the SolverFactory class. Excluding the Statistics (provides statistics) and DimacsOutput (produces CNF files) solvers, all the other pre-defined solvers are fuzzed.

Element	Missed Instructions	Cov.
newParallel()	<div><div></div></div>	100%
newBestCurrentSolverConfiguration(DataStructureFactory)	<div><div></div></div>	100%
newMiniLearningHeapEZSimpNoRestarts()	<div><div></div></div>	100%
newRandomSolver()	<div><div></div></div>	100%
newGreedySolver()	<div><div></div></div>	100%
newMiniSATHeap(DataStructureFactory)	<div><div></div></div>	100%
newBest17()	<div><div></div></div>	100%
newBackjumping()	<div><div></div></div>	100%
newGlucose()	<div><div></div></div>	100%
newMiniLearning(DataStructureFactory, IOrder)	<div><div></div></div>	100%
newSATUNSAT()	<div><div></div></div>	100%
newSAT()	<div><div></div></div>	100%
newGlucose21()	<div><div></div></div>	100%
newUNSAT()	<div><div></div></div>	100%
newMiniLearningHeapRsatExpSimp()	<div><div></div></div>	100%
newDefaultAutoErasePhaseSaving()	<div><div></div></div>	100%
newDefaultMS21PhaseSaving()	<div><div></div></div>	100%
newMiniLearningHeapRsatExpSimpLuby()	<div><div></div></div>	100%
newMiniLearningHeapEZSimp()	<div><div></div></div>	100%
newMiniLearningHeapExpSimp()	<div><div></div></div>	100%
newMiniSATHeapEZSimp()	<div><div></div></div>	100%
newMiniSATHeapExpSimp()	<div><div></div></div>	100%
newSizeLCDS()	<div><div></div></div>	100%
createInstance()	<div><div></div></div>	100%
newNoSimplification()	<div><div></div></div>	100%
newMiniLearningHeap(DataStructureFactory)	<div><div></div></div>	100%
instance()	<div><div></div></div>	100%
newMiniLearningHeap()	<div><div></div></div>	100%
newBestWL()	<div><div></div></div>	100%
newMiniSATHeap()	<div><div></div></div>	100%
newDefault()	<div><div></div></div>	100%
defaultSolver()	<div><div></div></div>	100%
newMiniLearningHeapEZSimpLongRestarts()	<div><div></div></div>	0%
newMiniLearningHeapRsatExpSimpBiere()	<div><div></div></div>	0%
newConcise()	<div><div></div></div>	0%
newMinOneSolver()	<div><div></div></div>	0%
newAgeLCDS()	<div><div></div></div>	0%
newActivityLCDS()	<div><div></div></div>	0%
newBestHT()	<div><div></div></div>	0%
newDimacsOutput()	<div><div></div></div>	0%
newStatistics()	<div><div></div></div>	0%
newLight()	<div><div></div></div>	0%
lightSolver()	<div><div></div></div>	0%
Total	101 of 547	81%

Figure 12: SolverFactory Coverage Report.

Figure 13 shows the coverage report of the different data structures and Figure 14 shows the coverage report for the order and phase heuristics. The `AbstractCardinalityDataStructure`, `AbstractDataStructureFactory`, and `AbstractPhaserecordingSelectionStrategy` classes are not actual configurations but abstract classes with method definitions [12]. It can be seen that the fuzzer covered all of the offered features in these categories.











Element	Missed Instructions	Cov.
AbstractCardinalityDataStructure		100%
MixedDataStructureDanielWL		81%
MixedDataStructureDanielWLConciseBinary		81%
MixedDataStructureSingleWL		75%
MixedDataStructureDanielHT		74%
ClausalDataStructureWL		73%
CardinalityDataStructure		52%
CardinalityDataStructureYanMax		51%
CardinalityDataStructureYanMin		51%
AbstractDataStructureFactory		43%
Total	205 of 707	71%

Figure 13: Data Structures Coverage Report.
















Element	Missed Instructions	Cov.
PureOrder		93%
ActivityBasedVariableComparator		90%
PhaseCachingAutoEraseStrategy		89%
RSATLastLearnedClausesPhaseSelectionStrategy		89%
AbstractPhaserecordingSelectionStrategy		85%
PhaseInLastLearnedClauseSelectionStrategy		83%
RSATPhaseSelectionStrategy		83%
VarOrderHeap		82%
RandomLiteralSelectionStrategy		81%
NaturalStaticOrder		76%
PositiveLiteralSelectionStrategy		75%
NegativeLiteralSelectionStrategy		75%
RandomWalkDecorator		71%
UserFixedPhaseSelectionStrategy		60%
SolutionPhaseSelectionStrategy		58%

Figure 14: Order and Phase Heuristics Coverage Report.

Figure 15 and Figure 16 show the coverage report for the learning and restart configurations respectively. AbstractLearning is not an actual configuration but an abstract class [12]. Both folders have high coverage (86% and 91%). The reason options have such high coverage is that they are used by the pre-defined solvers as well as when manually configuring them.










Element	Missed Instructions	Cov.
LimitedLearning		100%
NoLearningButHeuristics		100%
MiniSATLearning		92%
AbstractLearning		86%
PercentLengthLearning		85%
NoLearningNoHeuristics		81%
FixedLengthLearning		79%
ClauseOnlyLearning		75%
ActiveLearning		73%
Total	49 of 360	86%

Figure 15: Learning Strategies Coverage Report.














Element	Missed Instructions	Cov.
ArminRestarts		93%
EMARestarts		97%
FixedPeriodRestarts		74%
Glucose21Restarts		98%
LubyRestarts		89%
MiniSATRestarts		88%
NoRestarts		66%
Total	41 of 501	91%

Figure 16: Restart Strategies Coverage Report.

Figure 17 shows the code coverage report for the two internal enumerator classes, SearchEnumeratorListener and SolutionFoundListener [12]. The coverage for the onSolutionFound methods is 0 because I overwrite those methods to only increment the counter in the listener I attach to the solver when enumeration fuzzing [39].

Element	Missed Instructions	Cov.
solutionFound(int[], RandomAccessModel)		100%
SearchEnumeratorListener(SolutionFoundListener)		100%
init(ISolverService)		100%
getNumberOfSolutionFound()		100%
end(Lbool)		75%
static {...}		75%
Total	6 of 67	91%





Element	Missed Instructions	Cov.
static {...}		100%
onUnsatTermination()		100%
onSolutionFound(int[])		0%
onSolutionFound(IVecInt)		0%
Total	2 of 8	75%

Figure 17: Internal Enumerator classes, SearchEnumeratorListener and SolutionFoundListener, Coverage Report.

Figure 18 shows the coverage report for the external enumerator class, ModelIterator [12]. The only methods used during enumeration fuzzing are the isSatisfiable to find the models (in a loop) and the numberOfModelsFoundSoFar to return the number of models found at the end of execution, which is why the rest of the methods have 0 coverage.













Element	Missed Instructions	Cov.
model()		100%
isSatisfiable()		100%
ModelIterator(ISolver, long)		100%
ModelIterator(ISolver)		100%
numberOfModelsFoundSoFar()		100%
primeImplicant()		0%
isSatisfiable(IVecInt)		0%
clearBlockingClauses()		0%
reset()		0%
setBound(long)		0%
isSatisfiable(boolean)		0%
getBound()		0%
Total	110 of 184	40%

Figure 18: External Enumerator cClass, ModelIterator, Coverage Report.

5.2 Bugs Identified from API Fuzzing

The fuzzer was run with enabled assertions and identified 3 bugs in Sat4j. It also helped pinpoint the reason behind the internal enumerator’s wrong result [38]. After adding proof-checking it was able to detect 2 additional bugs. Some issues were easily triggered by running the fuzzer with a few iterations. Others needed to run the fuzzer overnight on multiple threads. This corroborates the observation that running more problems is more important than running fewer problems longer.

5.2.1 PureOrder Heuristics

The fuzzer detected a “Null Pointer Exception” in the PureOrder heuristic when checking that there is a watch connected to the literal being analyzed during the decision selection procedure [41]. The concept of data structures with watches is presented in Section 2.1.2. The trace that caused the error had 857 lines where all options were set and 3 incremental calls were made to the solver. To make it easier to analyze I put the trace through delta debugging and the reduced trace that helped identify the bug is shown in Figure 19. The original literals were -264 and -171 but were manually renamed to -3 and -2 because the renaming done by the DeltaDebugger [39] to -2 and -1 did not preserve the error.

Trace

1 init
3 Order : PureOrder/period=16
756 addClause -3 -2
857 solve

Figure 19: Trace that identified the PureOrder heuristic bug [41].

After analyzing the trace and the error stack I found that the error was in the PureOrder implementation. The issue arose because the variables did not belong to the formula, so the watches were not initialized. The fix was to first check that the variable is used in the formula before anything else. The trace proved to be useful for regression testing as well. After fetching the fix from the Sat4j repository [12] it was easy to verify that the fix worked by executing the TraceRunner [39] with the trace and checking if it was still throwing an error.

5.2.2 MinOne Solver

The fuzzer found a bug when applying MinOneDecorator with a solver allowing satisfied clauses simplifications [42]. There were traces with two different error types, namely “Array Index Out Of Bounds Exception” and “No Such Element Exception” but the underlying issue ended up being the same. One of the analyzed traces had 233 lines with 2 incremental calls made to the solver. After delta debugging the reduced trace is shown in Figure 20.

Trace
1 init
2 using solver MinOneSolver
3 DB simplification
44 addClause 2 4
73 addClause 1
84 addClause 4 3
136 solve
233 solve

Figure 20: Trace that identified the MinOne solver bug [42].

There were clauses added to the formula between the *solve* calls made to the solver but they did not affect the bug. As explained by Prof. Dr. Le Berre, in the case of MinOneDecorator, if the solver allows for constraints to be removed then there are cases where the constraints added for minimizing the model could also be removed.

5.2.3 Internal Enumerator

As mentioned in Section 4.1.4 the motivation behind adding support for internal and external enumerators in the fuzzer was the existing bug reported by users regarding incorrect results from the internal enumerator [38].

One of the analyzed traces produced 9 models from the internal enumerator and 8 from the external enumerator. The reduced file from delta debugging, presented in Figure 21, produced 65 models from the internal enumerator and 2 from the external enumerator. It is easy to deduct from looking at the trace that the models are $1 - 2$ $3 - 5 - 7$ and $1 - 2$ $3 - 5$ 7 . The clauses of this trace were added as a CNF file in the Sat4j repository [12] and used in a test method to test potential fixes for this bug and provide regression testing in the future.

Trace
1 init
12 addClause 3
13 addClause -7 2 -5
14 addClause -3 2 1
26 addClause -1 -5 -3
31 addClause -2
41 enumerating

Figure 21: Trace that helped narrow down the internal enumerator bug [38].

The issue, as Prof. Dr. Le Berre explained, was due to unit clauses being forgotten after a backjump. It was needed to backtrack to the decision level 0 and since the solver did not store that information a hack was used to look at the trail and backtrack to the place where it met the first propagated literal. The internal enumerator did not work properly for the example in Figure 21 because there was a unit propagation at decision level 0.

5.2.4 Head/Tail Data Structure

Another issue was discovered due to enumeration fuzzing when learning an empty clause [43]. If the empty clause is not added to the formula, the enumeration process can not end with an UNSAT answer. However, creating an empty clause using the Head/Tail data structures should not be possible. The data structure factory did not prevent that but it should have.

The trace that identified this is shown in Figure 22. The models found from the internal enumerator are $1 \ 2$, $1 \ -2$, and -1 . When trying to add -1 to the formula an “Assertion Error” was thrown from the solver. Though in the trace the `CardinalityDataStructureYanMin` data structure is used, the error is also present for `CardinalityDataStructureYanMax`, `CardinalityDataStructure`, `ClausalDataStructureWL`, `MixedDataStructureDanielHT`, and `MixedDataStructureSingleWL` data structures [12]. A unit test was created with all the H/T data structures to verify the fix and provide regression testing for this issue.

Trace

1 init
2 Data Structure Factory : `CardinalityDataStructureYanMin`
10 addClause -1 -2
25 enumerating

Figure 22: Trace that identified the HT data structure bug [43].

5.3 Bugs Identified from Proof-Checking

Proof-checking proved to be a useful addition to the API fuzzer capabilities. However, adding support for it was not without issues. The first set of requirements was clarified in a meeting with Prof. Dr. Le Berre based on the IDRUP format presented by Fazekas et al. [10]. The iCNF file is generated by the fuzzer and implemented by me and the IDRUP proof is generated by the connected listener to the solver and implemented by Prof. Dr. Le Berre [44]. After the needed changes were finalized I started using the new proof format and the created files to do proof-checking with IDRUP-CHECK [11].

The first issue encountered was a “Failed Lemma Implication Check Exception” thrown by the proof-checker. One of the traces and its proof after reduction is shown in Figure 23 where the error is thrown when learned clause 1 6 0 is reached. At first, I assumed that this reflected an issue in the solver learning strategy because, with other learning strategies, there was another clause learned first, 1 5 6 0.

Trace	IDRUP
1 init	p idrup
5 Learning Strategy : ActiveLearning/percent=0.92676	i -5 3 6 0
22 addClause -5 3 6	i 5 4 0
31 addClause 5 4	i -3 -5 0
96 addClause -3 -5	i 5 -2 0
131 addClause 5 -2	i -1 0
139 addClause -1	i -4 6 2 0
155 addClause -4 6 2	q 0
158 solve	l 6 0
	s SATISFIABLE
	m -1 -2 -3 4 5 6 0

Figure 23: Trace that identified the ignored clauses when generating IDRUP proof [44].

However, this error started appearing when using the default solver as well. After analyzing the issue and discussing it with Prof. Dr. Le Berre and Prof. Biere it was discovered that it was an issue with the proof generation. Not all constraints were being added to the solver, some were used for conflict analysis but not unit propagation so they were not considered as learned clauses. A fix was needed to output these discarded learnt clauses in the IDRUP proof.

5.3.1 DB Simplification

With proof-checking it was discovered that when DB simplification is allowed the solver would sometimes provide wrong results [45]. Figure 24 shows the reduced trace and the produced IDRUP proof that identified the issue. On the first search, the solver finds the correct model $-1 -2$. After adding the unit clause 2 the solver returns an incorrect model $-1 2$ instead of the correct UNSAT answer. This is found by the IDRUP-CHECK [11] which after comparing the iCNF and IDRUP files returns the error that the model $m -1 2 0$ does not satisfy the input clause $i -2 1 0$.

Trace	IDRUP
1 init	p idrup
11 DB simplification	i -2 1 0
46 addClause -2 1	i -1 0
269 addClause -1	q 0
291 solve	s SATISFIABLE
500 addClause 2	m -1 -2 0
1238 solve	i 2 0
	q 0
	s SATISFIABLE
	m -1 2 0

Figure 24: Trace that identified the DB simplification bug [45].

DB simplification also caused an enumeration bug where internal and external enumerators would differ by one. Figure 25 shows a reduced trace this issue was encountered on. After debugging I found that the internal enumerator returned only one model 1 2 while the external enumerator returned two models 1 2 and -1 2. The external enumerator had one additional model that was incorrect because -1 2 was not possible due to the third clause $2 -1$. The underlying issue was due to a missing check when enqueueing learned literals, thus unsatisfiability with unit clauses was not detected. Again a unit test was created with the two examples shown to verify the fix.

Trace
1 init
3 DB simplification
15 addClause 1 2
16 addClause 1 -2
21 addClause 2 -1
29 enumerating

Figure 25: Trace that identified the DB simplification bug when enumerating [45].

5.3.2 ExpensiveSimplificationWLOnly Learned Clauses

Another issue was found by the proof-checker when using the expensiveSimplification-WLOnly feature [46]. The trace that threw the “Lemma Implication Check Failed Exception” was 391 lines long, had all the options set, and made 2 incremental calls to the solver. Figure 26 shows the reduced trace after delta debugging. IDRUP-CHECK [11], when passed the produced IDRUP and iCNF files, would throw an error when reaching line 1 3 0. The correct learned clause should have been 1 3 4 0 which is learned by all other simplification techniques.

The issue was that `expensiveSimplificationWLOnly` simplification was supposed to be used only with the watched literal data structures, where the first literal is satisfied when the clause is a reason. In binary clause data structures, this is not enforced. As Prof. Dr. Le Berre explained, there were two fixes for this: enforcing the use of `expensiveSimplificationWLOnly` only for WL data structures (i.e. failing if a reason is not of that type) or asking the constraint the index to use for probing the first literal (0 by default, 1 for watched literals data structures). The second option was implemented and a unit test was created.

Trace	IDRUP
1 init	p idrup
7 Simplification Type : <code>expensiveSimplificationWLOnly</code>	i 3 -2 0
329 addClause 3 -2	i 1 -4 0
350 addClause 1 -4	i 4 3 2 0
357 addClause 4 3 2	q 0
391 solve	l 3 0
	s SATISFIABLE
	m -1 -2 3 -4 0

Figure 26: Trace that identified the `expensiveSimplificationWLOnly` bug [46].

5.3.3 MinOne Solver

Section 5.2.2 shows how the fuzzer discovered an issue with `MinOneSolver` and how it was fixed. However, it took too long for the `TraceRunner` [39] to finish executing one of the original traces during regression testing so a new problem was found when looking at the generated IDRUP proof [42]. The first search found a minimal model but the second search kept repeating the first found model. The IDRUP proof would reach 80,000+ (sometimes up to 200,000) lines before finishing with a SAT answer and there was only one model found during the second iteration.

The issue was not happening for the reduced traces, just the original ones. Because the solver did not technically throw an error it was not possible to put the trace through delta debugging to isolate the error-inducing input.

Prof. Dr. Le Berre stated that there are two loops when using an iterator on MinOne: one to get a model with a minimal number of ones and one to iterate over those models. Execution finishes successfully when satisfied clause deletion (DB simplification) is disabled. Once the first loop is applied, the minimization finishes with a cardinality constraint which propagates unit clauses. Those unit clauses then remove some satisfied clauses from the original formula. If the goal is to find only one minimal model then this is correct. However, if an additional loop is added to iterate over all models, some clauses have disappeared so the solver can no longer find the right models.

The solver should not get the same model again and again, it should have different ones but not necessarily models of the original formula because some constraints have been deleted. This issue was still open when this thesis was submitted [42].

6 Conclusions

This thesis aimed to increase the trust in Sat4j, the only competitive SAT solver implemented in Java. Sat4j has been in use for years and has been heavily tested by users so I was uncertain if any bugs remained to be found. Since fuzzing and proof-checking have been proven effective in identifying issues in SAT solvers [6, 8] it was decided to implement a model-based API fuzzing framework with integrated incremental proof-checking as presented in Chapter 3. These approaches have been previously implemented and used for C and C++ solvers, but to my knowledge, this is the first time they have been applied to a Java-based solver.

Chapter 5 presents the results of this work. The issues found by the fuzzer include a “Null Pointer Exception” when using the PureOrder decision heuristic [41], “Array Out Of Bounds Exception” from the MinOneSolver decorator [42], and “Assertion Errors” from the Head/Tail data structures when enumerating [43]. The fuzzer also helped fix known bugs such as the internal enumerator issue reported by users by providing reduced traces where the root cause was easier to identify [38].

IDRUP-CHECK was used as an external proof-checker [11]. Sat4j did not support incremental proof generation when I started working on this thesis but, in collaboration with Prof. Dr. Le Berre, it was added during the course of this project. Proof-checking was able to detect incorrect results returned by the solver when using DB simplification [45] and inconsistencies during reasoning when using the expensiveSimplificationWLOnly technique [46].

After the above-mentioned findings were fixed the fuzzer was initiated 1,558,623 times across multiple cores in parallel with 5 traces per run and a 10-second timeout for proof-checking. The fuzzer generated about 7.8 million seeds and did not find any new errors, indicating that all the bugs detectable from fuzzing and proof-checking have already been addressed.

Though Sat4j is regarded as an error-free solver due to its extensive testing over the years the fuzzer was still able to detect bugs, showing that the thesis successfully met its objective. These findings highlight the importance of fuzz testing, especially API fuzzing, and proof-checking when it comes to SAT solvers. The incremental approach, though used before in fuzzing, has only recently become available for proof-checking and has already proven useful [10].

The tool however has its limitations and is not able to detect every kind of bug. The second MinOneSolver bug presented in Section 5.3.3 was not found by the fuzzer but from manually analyzing the IDRUP proof it generated. Additionally, Prof. Dr. Le Berre discovered another issue when investigating the Head/Tail data structure bug (Section 5.2.4) where enumerators do not work correctly if the solver is empty.

Several opportunities exist for future research and development in this area. First, the current fuzzing framework could benefit from improvements in input generation techniques. Incorporating advanced algorithms, such as machine learning-based fuzzing, could help create more diverse and intelligent test cases, potentially uncovering deeper bugs in Sat4j's API. Additionally, applying the API fuzzing framework to other SAT solvers could yield valuable comparative insights, it could evaluate the effectiveness of different solvers under similar testing conditions and explore generalizations of the fuzzing technique. Extending this approach to PB, MaxSAT, or Parallel solvers available in Sat4j could also yield interesting results.

7 Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor, Prof. Dr. Armin Biere, for their guidance and insight throughout this project. Their expertise was invaluable, and I am deeply grateful for the opportunity to learn and grow under their mentorship.

I am very grateful to Prof. Dr. Daniel Le Berre for their collaboration on this project and for being consistently available to answer my questions. Their enthusiastic participation in our meetings and invaluable advice have been a tremendous source of support throughout this journey.

I would also like to thank my advisors, Dr. Mathias Fleury and Tobias Paxian, for their valuable feedback and suggestions, which helped shape this work into its final form. Their constructive criticism and support were crucial in refining and improving the project and the thesis.

Finally, I would like to thank my family and friends for their endless endorsement and encouragement. To my parents, Elida and Shkëlqim, and my brother, Kildi, thank you for your patience and understanding during the long hours and challenging times. Your belief in me was a constant source of motivation.

Bibliography

- [1] S. A. Cook, “The complexity of theorem-proving procedures,” in *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pp. 143–152, 2023.
- [2] J. Marques-Silva, “Practical applications of boolean satisfiability,” in *2008 9th International Workshop on Discrete Event Systems*, pp. 74–80, IEEE, 2008.
- [3] S. Wieringa *et al.*, “Incremental satisfiability solving and its applications,” 2014.
- [4] P. Godefroid, “Random testing for security: blackbox vs. whitebox fuzzing,” in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pp. 1–1, 2007.
- [5] “Sat competitions.” <https://satcompetition.github.io/>, 2024.
- [6] R. Brummayer, F. Lonsing, and A. Biere, “Automated testing and debugging of sat and qbf solvers,” in *Theory and Applications of Satisfiability Testing–SAT 2010: 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 13*, pp. 44–57, Springer, 2010.
- [7] R. Brummayer and A. Biere, “Fuzzing and delta-debugging smt solvers,” in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pp. 1–5, 2009.

- [8] A. Darbari, B. Fischer, and J. Marques-Silva, “Industrial-strength certified sat solving through verified sat proof checking,” in *Theoretical Aspects of Computing—ICTAC 2010: 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings* 7, pp. 260–274, Springer, 2010.
- [9] A. Van Gelder, “Verifying rup proofs of propositional unsatisfiability,” in *ISAIM*, 2008.
- [10] K. Fazekas, F. Pollitt, M. Fleury, and A. Biere, “Incremental proofs for bounded model checking,” in *27th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV’24), Kaiserslautern, Germany*, vol. 314, pp. 133–143, 2024.
- [11] A. Biere, “Github - arminbiere/idrup-check.” <https://github.com/arminbiere/idrup-check>, 2024.
- [12] “Sat4j repository.” <https://gitlab.ow2.org/sat4j/sat4j>, 2017.
- [13] D. Le Berre and A. Parrain, “The sat4j library, release 2.2,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2010.
- [14] S. Competitions, “Sat competition 2009: Benchmark submission guidelines,” *SAT Competition*, vol. 13, 2009.
- [15] S. Wieringa, M. Niemenmaa, and K. Heljanko, “Tarmo: A framework for parallelized bounded model checking,” *arXiv preprint arXiv:0912.2552*, 2009.
- [16] J. Marques-Silva, I. Lynce, and S. Malik, “Conflict-driven clause learning sat solvers,” in *Handbook of satisfiability*, pp. 133–182, ios Press, 2021.
- [17] A. Darwiche and K. Pipatsrisawat, “Complete algorithms,” in *Handbook of Satisfiability*, pp. 99–130, IOS press, 2009.

- [18] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern sat solvers,” in *Twenty-first international joint conference on artificial intelligence*, Citeseer, 2009.
- [19] J. Marques-Silva, “The impact of branching heuristics in propositional satisfiability algorithms,” in *Progress in Artificial Intelligence: 9th Portuguese Conference on Artificial Intelligence, EPIA’99 Évora, Portugal, September 21–24, 1999 Proceedings 9*, pp. 62–74, Springer, 1999.
- [20] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, “Understanding vids branching heuristics in conflict-driven clause-learning sat solvers,” in *Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17–19, 2015, Proceedings 11*, pp. 225–241, Springer, 2015.
- [21] K. Pipatsrisawat and A. Darwiche, “A lightweight component caching scheme for satisfiability solvers,” in *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28–31, 2007. Proceedings 10*, pp. 294–299, Springer, 2007.
- [22] A. Biere and A. Fröhlich, “Evaluating cdcl restart schemes,” *Proceedings of Pragmatics of SAT*, pp. 1–17, 2015.
- [23] A. Biere, “Picosat essentials,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, no. 2-4, pp. 75–97, 2008.
- [24] G. Audemard and L. Simon, “Refining restarts strategies for sat and unsat,” in *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8–12, 2012. Proceedings*, pp. 118–126, Springer, 2012.
- [25] M. J. Heule and A. Biere, “Proofs for satisfiability problems,” *All about Proofs, Proofs for all*, vol. 55, no. 1, pp. 1–22, 2015.

- [26] D. L. Berre, “Sat4j.” <http://www.sat4j.org/index.php>, 2020.
- [27] D. Le Berre and S. Roussel, “Sat4j 2.3. 2: on the fly solver configuration,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 8, no. 3-4, pp. 197–202, 2012.
- [28] N. Sörensson and A. Biere, “Minimizing learned clauses,” in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 237–243, Springer, 2009.
- [29] C. Artho, A. Biere, and M. Seidl, “Model-based testing for verification back-ends,” in *International Conference on Tests and Proofs*, pp. 39–55, Springer, 2013.
- [30] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 2123–2138, 2018.
- [31] “Java platform, standard edition 8 api specification.” <https://docs.oracle.com/javase/8/docs/api/overview-summary.html>, Jun 2024.
- [32] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Transactions on software engineering*, vol. 28, no. 2, pp. 183–200, 2002.
- [33] C. V. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto, “Modbat: A model-based api tester for event-driven systems,” in *Hardware and Software: Verification and Testing: 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings 9*, pp. 112–128, Springer, 2013.
- [34] A. Niemetz, M. Preiner, and A. Biere, “Model-based api testing for smt solvers,” in *SMT*, pp. 3–14, 2017.
- [35] A. Niemetz, M. Preiner, and C. Barrett, “Murxla: A modular and highly extensible api fuzzer for smt solvers,” in *International Conference on Computer Aided Verification*, pp. 92–106, Springer, 2022.

- [36] J. Scott, T. Sudula, H. Rehman, F. Mora, and V. Ganesh, “Banditfuzz: fuzzing smt solvers with multi-agent reinforcement learning,” in *International Symposium on Formal Methods*, pp. 103–121, Springer, 2021.
- [37] K. Fazekas, F. Pollitt, M. Fleury, and A. Biere, “Certifying incremental sat solving,” in *Proceedings of 25th Conference on Logic for Pro*, vol. 100, pp. 321–340, 2024.
- [38] “Internal model enumeration provides incorrect results on this specific input file.” <https://gitlab.ow2.org/sat4j/sat4j/-/issues/175>, Feb 2024.
- [39] I. Parruca, “Github - irisi99/sat4j-api-fuzzing.” <https://github.com/Irisi99/SAT4J-API-Fuzzing>, Aug 2024.
- [40] “Jacoco - java code coverage library.” <https://www.jacoco.org/jacoco/trunk/index.html>, 2024.
- [41] “Npe launched when using pureorder heuristics.” <https://gitlab.ow2.org/sat4j/sat4j/-/issues/179>, 2024.
- [42] “Issue with minone solver when removing satisfied clauses.” <https://gitlab.ow2.org/sat4j/sat4j/-/issues/184>, 2024.
- [43] “Ht data structure should not be called on empty clauses.” <https://gitlab.ow2.org/sat4j/sat4j/-/issues/182>, 2024.
- [44] “Implement idrup incremental proof format.” <https://gitlab.ow2.org/sat4j/sat4j/-/issues/176>, 2024.
- [45] “Issue when db simplification is allowed.” <https://gitlab.ow2.org/sat4j/sat4j/-/issues/181>, 2024.
- [46] “expensivesimplificationwlonly produces incorrect clauses.” <https://gitlab.ow2.org/sat4j/sat4j/-/issues/180>, 2024.

