Technical Report, April 2013

On CafeSat: A Modern SAT Solver for Scala

Régis Blanc

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

regis.blanc@epfl.ch

Abstract

We present CafeSat, a SAT solver written in the Scala programming language. CafeSat is a modern solver based on DPLL and featuring many state-of-the-art techniques and heuristics. It uses two-watched literals for Boolean constraint propagation, conflict-driven learning along with clause deletion, a restarting strategy, and the VSIDS heuristics for choosing the branching literal. CafeSat is both sound and complete.

In order to achieve reasonnable performances, low level and hand-tuned data structures are extensively used. We report experiments that show that significant speedup can be obtained from translating a high level algorithm written in a relatively idiomatic Scala style to a more C-like programming style. These experiments also illustrate the importance of modern techniques used by SAT solver. Finally, we evaluate CafeSat against the reference SAT solver on the JVM: Sat4j.

Categories and Subject Descriptors J6 [*Computer-Aided Engineering*]: Computer-Aided Design

General Terms Algorithms, Verification

Keywords Boolean satisfiability, constraint solving, verification.

1. Introduction

The Boolean satisfiability problem (SAT) is one of the most important problem in computer science. From a theoretical point of view, it is the first NP-complete problem. On the practical side, it is used as a target low level encoding for many applications. Since SAT solvers are well understood and have been engineered over many years, applications often choose to rely on them rather than developing a custom solver for the domain. One application domain is Electronic Design Automation (EDA), including automatic test generation [1] and logic synthesis[11]. Often those SAT solvers are also an important building block in the more general problem of constraint solving, and in particular as a basis for SMT solvers [7, 10].

In the Boolean satisfiability problem, one is given a set of clauses C, where each clause is a set of literals. A literal is either a propositional variable or the negation of a propositional variable. The goal is to find an assignment for the variables such that for each clause in C, at least one of the literal evaluates to true. More formally, the goal is to satisfy the following formula:

$$\bigwedge_{c \in C} \bigvee_{l \in c} l$$

This representation is called Conjunctive Normal Form (CNF).

In this paper, we present CafeSat, a complete SAT solver implemented in Scala. CafeSat is strongly inspired by MiniSat [6], a state-of-the-art SAT solver which is open source and written in C++. CafeSat implements many recent techniques present in modern SAT solvers. CafeSat is built around the DPLL scheme [3, 4]. Boolean constraint propagation (BCP) is implemented using the 2-watched literal scheme introduced by Chaff [13]. The branching heuristics is VSIDS, also introduced by Chaff. A key component of modern SAT solver is the conflict-driven clause learning [15, 16], allowing for long backtracking and restarting. CafeSat supports an efficient conflict analysis, with the 1UIP learning scheme and a clause minimization inspired from MiniSat.

Additionally, CafeSat exports an API for Scala. This enables some form of constraint programming in Scala, as alread promoted by Scala^{Z3} [9]. We illustrate its ease of use in Figure 1. The code implements a sudoku solver. A sudoku input is represented by a matrix of `Option[Int]`. We then generate nine variables for each entry, and generate all constraints required by the rules of sudoku. The constraints state how variables from the same rows, columns and blocks of a sudoku grid must relate to each other. Variables and constraints can be naturally manipulated as would any regular boolean expression in Scala.

Our library supports arbitrary boolean functions by implementing a structure preserving translation to CNF [14]. This transformation avoids the exponential blow up of the naive CNF transformation by introducing a fresh variable for each sub-formula and asserting the equivalence of the new variable with its corresponding sub-formula.

We believe CafeSat could have applications in the Scala world. The current release of the Scala compiler integrates a small SAT solver for the pattern matching engine. Complex

```
def solve(sudoku: Array[Array[Option[Int]]]) = {
  val vars = sudoku.map(_.map(_ ⇒ Array.fill(9)(boolVar())))
  val onePerEntry = vars.flatMap(row ⇒
                       row.map(vs ⇒ Or(vs:_*)))
  val uniqueInColumns = for(c ← 0 to 8; k ← 0 to 8;
                            r1 ← 0 to 7; r2 ← r1+1 to 8)
    yield !vars(r1)(c)(k) || !vars(r2)(c)(k)
  val uniqueInRows = for(r ← 0 to 8; k ← 0 to 8;
                         c1 ← 0 to 7; c2 ← c1+1 to 8)
    yield !vars(r)(c1)(k) || !vars(r)(c2)(k)
  val uniqueInGrid1 =
    for(k ← 0 to 8; i ← 0 to 2; j ← 0 to 2;
        r ← 0 to 2; c1 ← 0 to 1; c2 ← c1+1 to 2)
      yield !vars(3*i + r)(3*j + c1)(k) ||
            !vars(3*i + r)(3*j + c2)(k)
  val uniqueInGrid2 =
    for(k ← 0 to 8; i ← 0 to 2; j ← 0 to 2; r1 ← 0 to 2;
        c1 ← 0 to 2; c2 ← 0 to 2; r2 ← r1+1 to 2)
      yield !vars(3*i + r1)(3*j + c1)(k) ||
            !vars(3*i + r2)(3*j + c2)(k)
  val forcedEntries =
    for(r ← 0 to 8; c ← 0 to 8 if sudoku(r)(c) != None)
      yield Or(vars(r)(c)(sudoku(r)(c).get − 1))
  val allConstraints =
    onePerEntry ++ uniqueInColumns ++ uniqueInRows ++
    uniqueInGrid1 ++ uniqueInGrid2 ++ forcedEntries
  solve(And(allConstraints:_*))
}
```

Figure 1. Implementing a sudoku solver with CafeSat API.

systems on the JVM such as Eclipse also start to include SAT solving technology for their dependency management engines [12].

2. CafeSat

In this section, we present the architecture and features of CafeSat. We discuss the different heuristics implemented and also describe some of the data structures used. The solving component of CafeSat is currently about 1,300 lines of code. This does not include the API layer. CafeSat is open source and available on GitHub at the following URL:

`https://github.com/regb/scabolic`

This page contains instructions on how to build and run the system. Note that CafeSat is part of a bigger system, in development, intended to do constraint solving. In this sytem, CafeSat will play a central role.

Figure 2 depicts the high level DPLL procedure [3, 4]. The key steps are:

Decide. Choose a variable to branch on.

BCP. Deduce new assignment of variable.

Backtrack. Conflict analysis and backtrack.

```
var status = preprocess()
while(status == Unknown) {
  decide()

  var cont = true
  while(cont) {
    status = bcp()
    if(status == Conflict) {
      if(decisionLevel == 0)
        return Unsatisfiable
      else
        backtrack()
    } else if(status == Satisfiable)
      return Satisfiable
    else
      cont = false
  }
}
```

Figure 2. The classical DPLL procedure.

In general, we avoid recursion and try to use iterative constructs as much as possible. We use native JVM types whenever possible. We rely on mutable data structures to avoid expensive heap allocations. In particular, we make extensive use of `Array` with primitive types such as `Int` and `Double`. Those types are handled well by the Scala compiler, which is able to map them to the native `int[]` and `double[]` on the JVM.

The input (CNF) formula contains a fixed number N of variables, and no further variables are introduced in the course of the algorithm. Thus, we can represent variables by integers from 0 to $N-1$. Many properties of variables such as their current assignment and their containing clauses can then be represented using `Array` where the indices represent the variable. This provides a very efficient $O(1)$ mapping relation. Literals are also represented as integers, with even numbers being positive variables and odd numbers being negative variables.

We now detail the important components of the SAT procedure.

2.1 Branching Decision

The choice of the branching literal is essential. A good decision can make tremendous difference in the search time. In general, the intuition is to choose the literal that appears the most frequently in unsatisfiable clauses. However, this requires some expensive bookkeeping. There is a need for a good trade-off with a cheap heuristic that is relatively useful.

In CafeSat, we rely on the VSIDS decision heuristic introduced initially by Chaff [13]. However, we implement the variation of the heuristic described in MiniSat [6]. We keep variables in a priority queue, sorted by their current VSIDS score. On a branching decision, we extract the maximum element of the queue that is not yet assigned. This is the branching literal.

We use a custom implementation of a priority queue that supports all operations in $O(\log N)$, including a delete by value of the variables (without any use of pointers). The trick is to take advantage of the fact that the values stored in the heap are integers from 0 to $N-1$, and maintain an inverse index to their current position in the heap. The heap is a simple binary heap built with an array. In fact, we store two arrays, one for variables and one for their corresponding score. Having two separate arrays seem to be more efficient than one array of tuples.

2.2 Boolean Constraint Propagation

The BCP procedure (also called unit propagation) is executed many times during the search procedure. It needs to be made as fast as possible. The SAT solver Chaff introduced in 2001 the 2-watched literals implementation technique, which is still the most popular today.

CafeSat implements the original technique as described by the Chaff paper. We implement a custom `LinkedList` to store the clauses that are currently watching a literal. An important feature of our implementation is the possibility to maintain a pointer to elements we wish to remove, so that a remove operation can be done in $O(1)$ while iterating over the clauses. This is a typical use case for the 2-watched literal, where we need to traverse all clauses that are currently watching the literal, find a new literal to watch, add the current clause to the watchers of the new literal while removing it from the previous one. All operations need to be very fast because they are done continuously on all unit propagation steps.

2.3 Clause Learning

In the original DPLL algorithm, the exhaustive search was explicit, setting each variable to true and false successively after exploring the subtree. A more recent technique consists in doing conflict analysis and then learning a clause before backtracking. The intuition is that this learnt clause is a reason why the search was not able to succeed in this branch. This learning scheme also enables the solver to do long backtracking, returning to the first literal choice that caused the clause to be unsatisfiable and not the most recent one.

In CafeSat, we implement a conflict analysis algorithm to learn new clauses. For this, we use the 1UIP learning scheme [16]. We also apply clause minimization as invented by MiniSat. We use a stack to store all assigned variable and maintain a history. We also store for each variable the clause (if any) responsible for its propagation. This implicitly stores the implication graph used in the conflict analysis.

2.4 Clause Deletion

Each new clause added to the problem set will make unit propagation slower. At some point, it could be beneficial to forget the least used clauses and only keep the most active ones.

We use an activity based heuristic similar to the one used for decision branching to select which clauses to keep and which ones to drop. We set a maximum size to our set of learnt clauses, and whenever we cross this threshold, we delete the clauses with the worst activity score. To ensure completeness and termination, we periodically increase this threshold.

Our current implementation simply stores a list of clauses and sorts them each time we need to remove the least active ones. We assume that clause deletion only happens after a certain number of conflicts, so it is not a very frequent operation. Besides, it could be cheaper to only sort the list each time it is needed, than to maintain the invariant in a priority queue for each operation.

2.5 Restarting Strategy

We use a restart strategy based on a starting interval that slowly grows over time. The starting interval is N which is the number of conflicts until a restart is triggered. A restart factor R will increase the interval after each restart. This increases in the restart interval guarantees completeness of the solver. In the current implementation, $N = 32$ and $R = 1.1$.

2.6 Preprocessing

Preprocessing consists in simplifying the formula before actually running the top level loop. There exists some very sophisticated preprocessing scheme [5]. In CafeSat, preprocessing is extremely lightweight and only detects trivially satisfied clause (those containing both a literal and its opposite) and clauses containing a single variable.

3. Experiments

We ran a set of experiments to evaluate the impact of various optimizations that have been implemented over the development of CafeSat. The goal is to give some insight on how incremental refinement of a basic SAT solver can lead to a relatively efficient complete solver. We selected a few important milestones in the development of CafeSat, and compared their performance on a set of standard benchmarks.

Our results are summarized in Table 1. The experiments have been run on an Intel core I5-2500K with 3.30GHz and 8 GiB of RAM. A timeout was set to 30 seconds. The running time is shown in seconds. The versions are organized from the most ancient to the most recent one, their description is as follows:

naive. Based on the straightforward implementation techniques using AST to represent formulas, and recursive functions along with pattern matching for DPLL and BCP.

counters. Uses specialized clauses. Each variable is associated with adjacency lists of clauses containing the vari-

Version	*naive*		*counters*		*conflict*		*2-watched*		*minimization*		*optimization*	
Benchmark	Succ.	Time	Succ.	Time	Succ.	Time	Succ.	Time	Succ.	Time	Succ.	Time
uf20	100	0.171	100	0.046	100	0.085	100	0.090	100	0.052	100	0.052
uf50	100	0.171	100	0.127	100	0.325	100	0.336	100	0.084	100	0.081
uuf50	100	0.507	100	0.179	100	0.658	100	0.701	100	0.111	100	0.095
uf75	100	3.948	100	0.444	100	1.170	100	1.320	100	3.138	100	0.122
uf100	30	27.05	99	4.006	91	7.567	93	5.844	100	0.225	100	0.183
uuf100	44	25.42	94	10.81	45	25.06	53	18.24	100	0.369	100	0.275
uf125	0	NA	55	18.73	43	20.07	52	18.02	100	0.393	100	0.317
uf200	0	NA	0	NA	7	28.30	7	28.48	60	6.688	100	2.131
uf250	0	NA	0	NA	0	NA	0	NA	22	25.46	64	16.01

Table 1. Benchmarking over versions of CafeSat.

able. It uses counters to quickly determine whether a clause becomes SAT or leads to a conflict.

conflict. Introduces conflict-driven search with clause learning. This is a standard architecture for modern SAT solver. However the implementation at this stage suffers from a lot of overhead.

2-watched. Implements the BCP based on 2-watched literals.

minimization. Focuses on a more efficient learning scheme. The conflict analysis is optimized and the clause learnt is minimized. It also introduces clause deletion.

optimization. Applies many low level optimizations. A consistent effort is invested in avoiding object allocation as much as possible, and overhead is reduced thanks to the use of native `Array` with `Int` as much as possible. We implemented dedicated heap and stack data structures, as well as a linked list optimized for our 2-watched literal implementation.

The benchmarks are taken from SATLIB [8]. We focus on uniform random 3-SAT instances, as SATLIB provides a good number of them for many different sizes. Thus, we are able to find benchmarks that are solvable even with the very first versions, and this results in better comparisons.

From these results we can see that the *naive* version is able to solve relatively small problems and has little overhead. On the other hand, it is unable to solve any problem of consequent size. The introduction of the conflict analysis (version *conflict*) had actually a lot of overhead in the analysis of the conflict and thus did not bring any performance improvement. The key step is the optimization of this conflict analysis (version *minimization*), this diminishes the overhead on the conflict analysis, thus reducing time spent in each iteration, and minimizing the learning clause. Smaller clause implies more triggers for unit propagation and a better pruning of the search space.

It is somewhat surprising that the addition of the 2-watched literal scheme has little effect on the efficiency of the solver. The implementation at that time was based on Scala `List` standard library. The *optimization* version introduces dedicated data structure to maintain watcher clauses. These results show that without a carefully crafted implementation, even smart optimizations do not always improve performance.

Benchmark	CafeSat		Sat4j	
	% Suc.	Time (s)	% Suc.	Time (s)
uf50	100	0.0014	100	0.0008
uf100	100	0.0040	100	0.0032
uuf100	100	0.0069	100	0.0063
uf125	100	0.0136	100	0.0119
uf200	100	0.5526	100	0.2510
uf250	63	4.5972	100	2.3389
bmc	92	3.9982	100	1.4567

Table 2. CafeSat vs Sat4j: Showdown.

To give some perspective on the performance of CafeSat, we also ran some comparison with a reference SAT solver. We chose Sat4j [2] as it is a fast SAT solver written for the JVM. CafeSat (as well as Sat4j) is currently unable to compete with SAT solvers written in C or C++. Thus, our short term goal will be to match the speed of Sat4j.

The experiments are summarized in Table 2 with the percentage of successes and average time. We set a timeout of 20 seconds. The average time is computed by considering only instances that have not timeouted. We used the most recent version of CafeSat and turned off the restarting strategy. We compared with Sat4j version 2.3.3, which, as of this writing, is the most recent version available. We use a warm-up technique for the JVM, consisting in solving the first benchmark from the set 3 times before starting the timer. The **bmc** benchmarks are formulas generated by a model checker on industrial instances. They are also standard problem from SATLIB. They contain up to about 300,000 clauses.

Our solver is competitive with Sat4j on the instances of medium sizes, however it is still a bit slow on the biggest instances. That CafeSat is slower than Sat4j should not come as a shock. Sat4j has been under development for more than

5 years and is considered to be the best SAT solver available on the JVM.

4. Conclusion

We presented CafeSat, a modern SAT solver written in Scala. CafeSat offers solid performance and provides Scala programmers with a library for constraint programming. This library makes access to SAT solving capabilities very easy in the Scala ecosystem offering a native solution with the usual feeling of a Scala DSL.

CafeSat is a DPLL based SAT solver. It is both sound and complete. It integrates many state-of-the-art techniques and heuristics that are currently in use in some of the most popular SAT solvers.

We used an extensive set of standard benchmarks to evaluate the improvement of CafeSat over time. These results give some insight on the importance of good heuristics and careful hacking. We also compared CafeSat to Sat4j, and despite Sat4j being superior, our new solver shows some promising initial results.

We plan to build a complete constraint solver on top of CafeSat. To that end, we will extend CafeSat with incremental SAT solving. We also aim to provide a constraint programming API to use our extended system. We hope to make CafeSat a solid infrastructure on which Scala programmers can build.

Acknowledgments

The author would like to thank Viktor Kuncak and Alexandre Duc for comments on this report, as well as Philippe Suter for precious advices on implementation details.

References

[1] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue. Generating Combinatorial Test Cases by Efficient SAT Encodings Suitable for CDCL SAT Solvers. In C. Fermller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[2] D. L. Berre and A. Parrain. The Sat4j Library, Release 2.2. *JSAT*, 7(2-3), 2010.

[3] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3), July 1960. ISSN 0004-5411.

[4] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7), July 1962. ISSN 0001-0782.

[5] N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, SAT'05, Berlin, Heidelberg, 2005. Springer-Verlag.

[6] N. En and N. Srensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004.

[7] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004.

[8] H. H. Hoos and T. Sttzle. SATLIB: An Online Resource for Research on SAT. IOS Press, 2000.

[9] A. S. Köksal, V. Kuncak, and P. Suter. Scala to the Power of Z3: Integrating SMT and Programming. In *CADE*, 2011.

[10] S. Krstić and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *Proceedings of the 6th international symposium on Frontiers of Combining Systems*, FroCoS '07, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] W. Kunz and D. Stoffel. *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[12] D. Le Berre and P. Rapicault. Dependency Management for the Eclipse Ecosystem: Eclipse p2, Metadata and Resolution. In *Proceedings of the 1st international workshop on Open component ecosystems*, IWOCE '09, New York, NY, USA, 2009. ACM.

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, New York, NY, USA, 2001. ACM.

[14] D. A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *J. Symb. Comput.*, 2(3), Sept. 1986. ISSN 0747-7171.

[15] J. a. P. M. Silva and K. A. Sakallah. GRASP: a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, Washington, DC, USA, 1996. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=244522.244560`.

[16] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, Piscataway, NJ, USA, 2001. IEEE Press.