Flexible Implementation of SAT solvers *

Filip Marić

Faculty of Mathematics, University of Belgrade,
filip@matf.bg.ac.rs

Abstract. We propose a flexible and modular object-oriented architecture for DPLL-based SAT solvers. Its main feature is a clear separation of the core DPLL algorithm from various external modules (e.g., heuristic parts of the solver, user interface, unsatisfiability proof logging). This allows co-existence of different techniques (especially heuristic policies) within the same solver making the solver more powerful, adaptive and also suitable for different experimental research in the field of SAT. The architecture is based on abstract state transition systems for SAT and on well known object-oriented design patterns. The proposed architecture serves as a basis of our SAT solver ArgoSAT. Contribution of this work is in software design of SAT solvers and it is aimed primarily to SAT solver developers.

1 Introduction

The propositional satisfiability problem (SAT) is the problem of deciding if there is a truth assignment under which a given propositional formula (in conjunctive normal form) evaluates to true. It is a canonical NP-complete problem [Coo71] and has important practical applications in electronic design automation, software and hardware verification, artificial intelligence, and operations research, etc. The majority of the state-of-the-art complete SAT solvers are based on the Davis-Putnam-Logemann-Loveland, or DPLL [DP60,DLL62].

Spectacular improvements in the performance of DPLL-based SAT solvers have been achieved in the last decade. Three different layers made these improvements possible: (i) high-level conceptual enhancements of the original DPLL procedure aimed at reducing the amount of explored search space (e.g., *backjumping*, *conflict-driven lemma learning*, and *restarts*), (ii) low-level implementation techniques (e.g., smart data structures, most notably the *two-watch literals* scheme for unit propagation, memory management techniques), and (iii) heuristic components of solvers (e.g., policies for literal selection, restarting, clause database management). However, although it became possible to decide satisfiability of industrial SAT problems with tens of thousands of variables and millions of clauses, software design of modern SAT solvers is often neglected and it can be improved in many ways. This paper is an attempt to emphasize the importance of this aspect of SAT solver implementation and to propose some solutions.

* This work was partially supported by Serbian Ministry of Science grant 144030.

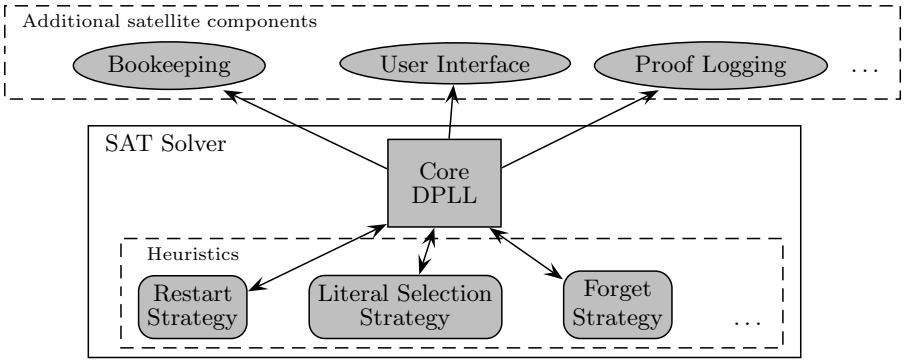

Fig. 1. The core DPLL and its satelite modules

Many of the individual ideas employed have already been present in available open-source solvers[1], but we try to summarize and describe them in an uniform way. We propose an architecture based on abstract state transition systems for SAT [NOT06,KG07]. It is formulated in an object-oriented setting and relies on well-known design patterns [GHJ+95]. The contribution of this work is in software design of SAT solvers, and not in SAT solving algorithms.

The main feature of our architecture is encapsulation of different SAT solving concepts into separate modules. Indeed, when the solver development is driven only by the quest for maximal speed (crucial for practical usability), if special attention is not paid to the software design, the code usually becomes "hard-wired" and monolithic. It usually implements a plethora of smart implementation tricks, but as a consequence becomes non-flexible and hard to modify. It is often the case that different layers (higher-level algorithms, lower-level implementation techniques and heuristics) are intermixed. A single function in the code can implement multiple SAT solving concepts, and conversely, implementation of a single concept (e.g., a heuristic) is often spread out in many functions in the code. In contrast, our approach tends to isolate different concepts present within a SAT solver into separate modules. We also make a clear and explicit distinction between three layers listed above. All this makes the code clear, easier to read, understand, maintain and to reason about. The most important step in this direction is the separation the core DPLL procedure from various satellite modules (e.g., heuristic components, user interface, bookkeeping, generating unsatisfiability proofs), as illustrated in Fig. 1.

As a consequence of the flexible architecture, it becomes possible to support co-existence of a number of different techniques (especially heuristics) within a single SAT solver and modification of existing and addition of new components can be easily performed. This can be beneficial for several reasons. First, fair comparisons between different techniques could be made, making the solver

[1] In author's opinion, Sat4J (`http://www.sat4j.org`) is the solver in which the most attention has been payed to the software design.

suitable for performing experimental research in the field of SAT. Next, a solver could choose between multiple policies and select the one that is most suitable for the input formula[2]. Namely, various heuristics often show different behavior on different benchmark families and it is rarely the case that one of them is universally superior[3].

This article is aimed primarly to SAT solver developers. The proposed architecture is implemented within our SAT solver ArgoSAT[4], and all code samples that will be shown are taken in verbatim from its C++ code. We will illustrate our architecture by giving descriptions of the implementation of some parts of the solver. However, due to the lack of space, many parts of the proposed architecture will be omitted. In the rest of the paper, familiarity with state-of-the-art SAT solving technology and the basics of object-oriented design is assumed.

2 Abstract State Transition Systems for SAT

During the last few years, two *abstract state transition systems* which model the DPLL-based SAT solvers and related SMT solvers have been published [NOT06,KG07]. These descriptions define the top-level architecture common for most state-of-the-art SAT solvers as mathematical objects that are easy to comprehend and that can be fruitfully reasoned about. Both systems are accompanied with pen-and-paper correctness and termination proofs. Although they succinctly and accurately capture all major aspects of the solvers' global operation, they are high level and far from the actual implementations. Many aspects not described by the rules have to be specified in order to get an effective implementation. The solver's behaviour is modelled by transitions between states which represent the values of its global variables. These include a set of clauses F and an assertion trail M. Transitions between states are performed only by using precisely defined transition rules. The solving process terminates when no more transition rules apply and a final state is reached.

The system presented in [KG07] gives a more detailed description about some phases of the solving process (particularly the conflict analysis phase) then the system from [NOT06], so we list its transition rules in Fig. 2. The rules have a guarded assignment form: above the line is the condition that enables the rule, below the line is the update to the state variables. Along with the *formula* F and the *trail* M, the state of the solver is characterized by the conflict analysis set C which is either a set of literals or the distinguished symbol no_cflct. The input to the system is an arbitrary set of clauses F_0, modeled by an initial state in which $F = F_0$, $M = [\,]$, and $C = no\_cflct$.

[2] Recent research [HHH+06,XHH+08,NMJ09] shows that it is possible to guess which of several given heuristics would give the best results for a given formula, based on its syntactical characteristics.

[3] For example, in [Hua07] MiniSat [ES04] and Berkmin [GN02] literal selection strategies are compared and it is concluded that for either of them there are benchmark families on which it performs better.

[4] The web page of ArgoSAT is `http://argo.matf.bg.ac.yu`

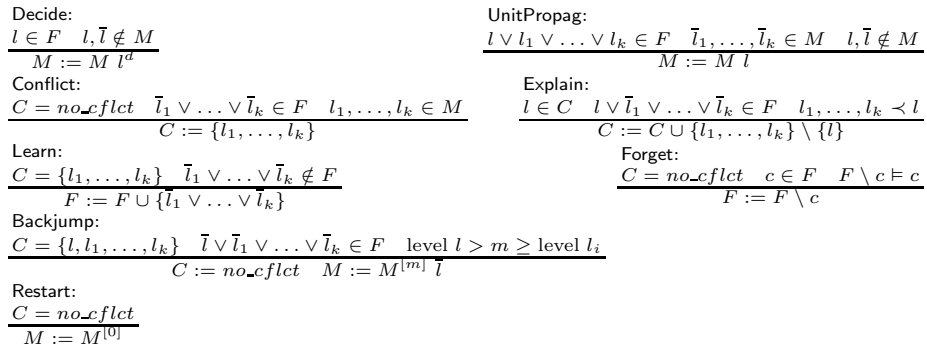

Fig. 2. Rules of DPLL as given in [KG07]

3 Core DPLL

Most of today's DPLL-based SAT solvers implicitly implement state transition systems for SAT. The architecture that we are proposing suggests that this relationship should be made explicit.

The *core DPLL component*, having precisely the functionality of an abstract state transition system for SAT, should be isolated within a SAT solver (in what follows, this component will be represented by the class `Solver`). Its interface should explicitly match all state transition rules (backjumping, learning, etc.) — both their applications and checking for their applicability, while its implementation should employ smart low-level techniques (such as the two-watch literal propagation) in order to make rule applications as efficient as possible. As an example, we list the method that applies the Decide rule (the used concepts of literal selection strategy and listeners will be explained in Sect. 4.1 and Sect. 3.1, and the rest of the code implements the rule effect):

```
void Solver::applyDecide() {
    Literal decisionLiteral = _literalSelectionStrategy->getLiteral();
    setReason(decisionLiteral, 0);
    assertLiteral(decisionLiteral, true);
    for (li = _listeners.begin(); li != _listeners.end(); li++)
      (*li)->onDecide(decisionLiteral);
}
```

This way, the most complex parts of a SAT solver are encapsulated within the implementation of the core component. If this component was trusted (or formally verified) to implement the transition rules correctly, then correctness of the whole solver would be guaranteed by the correctness of the underlying abstract state transition system (under the condition that heuristic components satisfy some additional, usually trivial, conditions).

A number of heuristics specify when and how the transition rules are applied, in a sense that they determine the order and the missing aspects of rule applications. All of them should be isolated from the core DPLL component, so

that they can be externally defined (as it is the case with the literal selection in the given example).

It is possible to adapt existing solvers and make them meet these requirements. However, this question has to be addressed separately for each specific SAT solver implementation and goes beyond the scope of this paper. A detailed description of a solver which is a rational reconstruction of MiniSat [ES04], following the rules of [KG07], is available in [Mar09].

3.1 Communication Between the Core DPLL and the Satellite Modules

Interface of the core DPLL component, must support communication between the core and all its satellite components.

First, the solver's internal state (the trail M and the set of clauses F, split into initial and learnt clauses) should be accessible (as read only):

```
class Solver {
public: ...
   const Trail& getTrail() const;
   const std::vector<const Clause*>& getInitialClauses () const;
   const std::vector<const Clause*>& getLearntClauses  () const;
};
```

Next, satellite components must be informed about all relevant actions that the solver performs. A convenient way to enable this is to use the *Observer* design pattern [GHJ+95]. In this case, every object interested in tracking solver's actions must be of a class that implements the `SolverListener` interface. The `SolverListener` interface directly corresponds to the transition rules of [KG07]:

```
class SolverListener {
 public:
  virtual void onDecide    (Literal l)                            {}
  virtual void onPropagate (Literal l, Clause* clause)            {}
  virtual void onBacktrack (Literal l)                            {}
  virtual void onConflict  (Clause* conflictClause)               {}
  virtual void onExplain   (Clause* C, Literal l, Clause* clause) {}
  virtual void onLearn     (Clause* clause)                       {}
  virtual void onForget    ()                                     {}
  virtual void onRestart   ()                                     {}
};
```

Listeners must register to the solver by calling its `addListener` method and this is usually done during their construction. Listeners can require to stop tracking the solver by calling its `removeListner` method and this is usually done during their destruction.[5]

[5] In further text, class constructors and destructor will usually be ommited because they implement trivial operations (e.g., registering listeners, resizing arrays).

```
class Solver {
public: ...
  void addListener    (SolverListener* listener) const;
  void removeListener (SolverListener* listener) const;
private: ...
  std::vector<SolverListener*> _listeners;
};
```

In all places in the core DPLL code corresponding to high-level rule applications, all registered listeners should be notified so they can take appropriate actions, as this is the case in the `applyDecide` function given on the page 4.

4 Heuristic Components of the Solver

In order to get an effective SAT solver implementation, heuristics that determine how and when are the abstract state transition rules (implemented by the core DPLL) applied, must be specified. Today's solvers feature a large number of heuristic components. As an illustration, we will discuss *literal selection strategy* and *restart strategy*. Similar implementation techniques can be also applied on other heuristic components (e.g., on the *forget strategy* which determines when to apply forgetting, *forget selection strategy* which determines which clauses to forget, *conflict analysis strategy* which determines the conflict analysis procedure). The communication between these heuristics and the core DPLL is bidirectional. Once these heuristic components are separated from the core DPLL procedure, the support for multiple variants of a heuristic within a same solver can be naturally achieved by using the *Strategy* design pattern [GHJ+95].

4.1 Literal selection

The only responsibility of a *Literal selection strategy* is to select a next literal for branching (i.e., for the application of the rule Decide):

```
class LiteralSelectionStrategy {
public:
   virtual Literal getLiteral() = 0;
};
```

The solver is associated with a literal selection strategy object:

```
class Solver {
public: ...
   void setLiteralSelectionStrategy(LiteralSelectionStrategy* strategy);
private: ...
   LiteralSelectionStrategy* _literalSelectionStrategy;
};
```

As shown in the `applyDecide` function given on the page 4, the strategy object is consulted whenever a new decision should be made.

Separate selection of variables and polarities. Most literal selection strategies first select the variable for branching and only afterwards select its polarity:

```
class LiteralSelectionStrategy_VariablePolarity :
   public LiteralSelectionStrategy {
public:
  Literal getLiteral() {
    Variable var = _variableSelectionStrategy->getVariable();
    Literal  lit = _polaritySelectionStrategy->getLiteral(var);
    return lit;
  }
private:
  VariableSelectionStrategy* _variableSelectionStrategy;
  PolaritySelectionStrategy* _polaritySelectionStrategy;
};

class VariableSelectionStrategy {
public:
   virtual Variable getVariable() = 0;
};

class PolaritySelectionStrategy {
public:
   virtual Literal getLiteral(Variable variable) = 0;
};
```

MiniSAT Variable Selection Strategy. As an example, we shall show how a very complex variable selection strategy like the one used in MiniSAT [ES04] is easily modelled within our framework. Notice that its implementation is completely separated from the core DPLL algorithm, and this design solution is our main contribution to this fragment of the solver. Also, this example shows how it is possible to make an explicit separation between a higher-level functionality of a heuristic and its lower-level implementation technique, and this hopefully makes the code much easier to comprehend.

The main idea behind the MiniSAT variable selection strategy is to give priority to variables that were involved in recent conflicts. This is achieved by assigning a numeric *activity score* to each variable. When choosing a variable to branch on, the one with the maximal activity score among all variables undefined in the trail is chosen. The scores dynamically change whenever variables are involved in conflicts. On each application of the rule Conflict and the rule Explain, the activity score for each variable of the clause involved is *bumped*. As the strategy tends to prioritize recent conflicts, on each application of Conflict, the activity of all variables is *decayed*. Notice that because this strategy must respond to solver's actions, it must implement the `SolverListener` interface.

In order to optimize finding variables with high activity scores, variables are kept in a *heap* data structure ordered by their activity scores. For efficiency reasons, the heap is updated "lazily", i.e., variables are pushed on to it each

time they are backtracked from the trail, but are not removed from it each time they get asserted. This "excess" variables on the heap are filtered out when the final decision variable is selected.

The activity scores are maintained by a separate class. It stores scores for all variables and its interface must enable bumping a score of a specified variable and decaying scores for all variables. It can be implemented in several different ways, and one efficient way will be discussed in the sequel.

```
class MinisatVariableSelectionStrategy :
         public VariableSelectionStrategy, public SolverListener {
public:...
    Variable getVariable() {
        Variable maxVar;
        do {
            maxVar = _activityHeap.pop_heap();
        } while (!(_solver.getTrail().isUndefVariable(maxVar)));
        return maxVar;
    }
    void onConflict(Clause* clause) {
        _activities.decayAll();
        bumpVariablesInClause(clause);
    }
    void onExplain(Clause* C, Literal literal, Clause* clause) {
        bumpVariablesInClause(clause);
    }
    void onBacktrack(Literal literal) {
        Variable variable = getVariable(literal);
        if (!_activityHeap.contains(variable))
            _activityHeap.push_heap(variable);
    }
private:
    void init() {
        for (Variable var = 0; var < _solver.numVars(); var++)
            _activityHeap.push_heap(var);
    }
    void bumpVariablesInClause(Clause* clause) {
        for (lit = clause->begin(); lit != clause->end(); lit++)
            bumpVariableActivity(getVariable(*lit));
    }
    void bumpVariableActivity(Variable variable) {
        _activities.bump(variable);
        if (_activityHeap.contains(variable))
            _activityHeap.increase(variable);
    }
    const Solver& _solver;
    Activities _activities;
    Heap<Variable, Activities::Comparator> _activityHeap;
};
```

The implementation of the `Activities` class can employ some smart low-level tricks introduced by MiniSAT. First, activity scores are represented as floating point numbers. For efficiency reasons, when the decay of all variable scores is requested, variable scores remain intact, but instead, only the *bump factor* is increased. In this way, the relative proportions of activity scores remain the same, but the implementation is, of course, much faster. However, there is one caveat: since the bump factor grows exponentially, overflow can occur and to prevent it, activity scores must be rescaled from time to time.

```
class Activities {
public:
  Activities(double bumpAmount, double decayFactor, size_t numVars)
    : _bumpAmount(bumpAmount), _decayFactor(decayFactor) {
        _activities.resize(numVars);
        _activities.assign(numVars, 0.0);
  }
  void bump(Variable variable) {
    _activities[variable] += _bumpAmount;
    if (_activities[variable] > MAX_ACTIVITY)
      rescaleActivities();
  }
  void decayAll() {
    _bumpAmount *= _decayFactor;
  }
  double getActivity(Variable variable) const {
     return _activities[variable];
  }
  ...
private:
  void rescaleActivities() {
    for (Variable var = 0; var < _activities.size(); var++)
           _activities[var] *= 1.0/MAX_ACTIVITY;
    _bumpAmount *= 1.0/MAX_ACTIVITY;
  }
  std::vector<double> _activities;
  double _bumpAmount;
  double _decayFactor;
};
```

Polarity caching. As an example of a non-trivial polarity selection strategy we will show implementation of the strategy known as *phase caching* [PD07]. A preferred polarity is assigned to each variable. Whenever a literal is asserted to the current assertion trail (either as a decision or as a propagated literal), its polarity defines the future preferred polarity of its variable. When a literal is removed from the trail (during backjumping or restarting) its preferred polarity is not changed.

```
class PolaritySelectionStrategyCaching :
   public PolaritySelectionStrategy, public SolverListener {
```

```
public:...
  Literal getLiteral(Variable variable) {
    return Literal(variable, _preferredPolarity[variable]);
  }
  void onAssert(Literal literal) {
    _preferredPolarity[getVariable(literal)] = isPositive(literal);
  }
private:
  std::vector<bool> _preferredPolarity;
};
```

Randomization as decoration. Some studies show that adding a small percentage of random decisions into literal selection heuristics can improve solving efficency [GSK98]. Since this can be done whatever variable selection strategy is used, it is convenient to implement is by using the *Decorator* design pattern [GHJ+95]:

```
class VariableSelectionStrategyRandomDecorator :
   public VariableSelectionStrategy {
public:...
    Variable getVariable() {
        if (randFloat() <= _percentRandom)
            return getRandomUndefinedVariable(); // make random decision
        else
            return _strategy->getVariable(); // fallback to default
    }
private:
    VariableSelectionStrategy* _strategy;
    float _percentRandom;
};
```

4.2 Restarting

The only responsibility of the *restart strategy* is to determine whether the solver should restart:

```
class RestartStrategy {
public:
   virtual bool shouldRestart() = 0;
};
```

Most restart strategies are based on conflict counting:

```
class RestartStrategyConflictCounting :
   public RestartStrategy, public SolverListener {
public:
  void init() {
    _numRestarts = 0; _numConflicts = 0;
    calculateConfilctsForFirstRestart();
  }
  void onConflict(Clause* conflictClause) {
```

```
    _numConflicts++;
  }
  void onRestart() {
    _numRestarts++; _numConflicts = 0;
    calculateConfilctsForNextRestart();
  }
  bool shouldRestart() {
    return _numConflicts >= _numConflictsForNextRestart;
  }
 protected:
  virtual void calculateConfilctsForFirstRestart() = 0;
  virtual void calculateConfilctsForNextRestart() = 0;
  size_t _numRestarts;
  size_t _numConflicts;
  size_t _numConflictsForNextRestart;
};
```

Only functions that specify number of conflicts for first, and for every next restart must be implemented in order to obtain a concrete restart strategy. MiniSat [ES04] uses the following strategy:

```
  void calculateConfilctsForFirstRestart() {
    _numConflictsForNextRestart = _numConflictsForFirstRestart;
  }
  void calculateConfilctsForNextRestart() {
    _numConflictsForNextRestart *= _restartInc;
  }
```

Picosat [Bie08] uses the following strategy:

```
  void calculateConfilctsForFirstRestart() {
    _numConflictsForNextRestart = _numConflictsForFirstRestart;
  }
  void calculateConfilctsForNextRestart() {
    if (_inner >= _outer) {
      _outer *= _restartInc; _inner = _numConflictsForFirstRestart;
    } else {
      _inner *= _restartInc;
    }
    _numConflictsForNextRestart = _inner;
  }
```

5 Additional Satellite Components

Once the core DPLL algorithm is encapsulated in a separate module with a clear and rich programming interface, it is possible to extend the basic SAT solver with many satellite modules implementing a wide range of different functionalities. In this section we just list some of them.

5.1 Bookkeeping

The statistics gathered during the solving process (e.g., the number of decisions, the number of conflicts, the current speed measured by the number of assertions per second) can offer deeper insights and improve the field of SAT research. The described architecture enables very easy implementation of bookkeeping of various statistics about the solving process, and their collecting can be performed by a satellite component.

5.2 User Interface

In order to give users some feedback about the state and the progress of the solving process, it is desirable to implement some kind of logging or output of different verbosity. With our architecture, these user interface components can be implemented independently from the SAT solver itself. As an example we show how to implement a simple character based output that helps tracking high-level transition rules that the solver applies:

```
class DotLoggingSolverListener : public SolverListener {
public:...
  void onAssert(Literal l) {
        if (_solver.getTrail().currentLevel() == 0)
           output("z");
  }
  void onDecide(Literal l) {
        if (_decisions++ == 1000)
           output("."), _decisions = 0;
  }
  void onRestart() { output("r\n"); }
  void onForget () { output("f");   }
};
```

If a more detailed information should be displayed, a graphical user interface (GUI) is preferred. Fig. 3 shows a graphical user interface of our solver ArgoSAT, also implemented on top of the basic solver without any need for making its modifications.

5.3 Proof Logging

Generating explicit proofs of unsatisfiability improves confidence in solvers results. Although there are several different standard proof formats, all of them can easily be implemented as satellite components within our framework. As an example, we sketch a generator for the *resolution-trace* format (used in the SAT competitions). Although there are some additional details that must be specified in order to get a correct proof logger, the main step is the logging of the Explain rule applications.

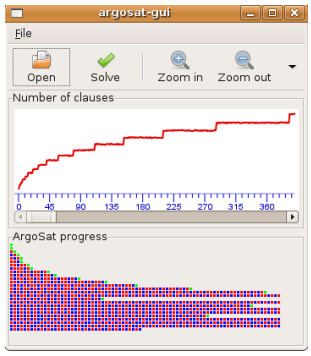

Fig. 3. ArgoSAT graphical user interface. The upper panel displays a graph showing the number of clauses remaining after each forget rule application, and the lower panel displays state transition rule applications (encoded by different colors).

```
class ResolutionTraceProofLogger {
public:...
   void onExplain(Clause* C, Literal l, Clause* clause) {
     dumpToFile(_solver.getC()->getID(), getVariable(l),
                C->getID(), clause->getID());
   }
};
```

6 Conclusions

In this paper, we advocated for the importance of the software design in the implementation of modern SAT solvers. We have described one object-oriented framework for the implementation of DPLL based SAT solvers, based on transition rules of [KG07]. Some of the proposed solutions are already present in available open-source SAT solvers, but we have extended them and presented them in an uniform way. The main contribution of our architecture is encapsulation of many different SAT solving concepts into separate modules and clear differentiation between high-level algorithms, heuristics, and low-level implementation techniques.

There is a trade-off between a nice code design and its efficiency. Namely, modularization introduces some runtime overhead when compared to a hard-coded variant of SAT solver, because of the increased communication between the core and the satellite components. However, experiments with our solver ArgoSAT show that this overhead is well below 3% on all benchmarks used for testing.

Solvers implementations significantly differ among themselves and, of course, it is not possible to specify one architecture suitable for all of them. However, we hope that the ideas described in this paper could help SAT solver developers

make flexible solvers, hopefully sharing some code and featuring multiple algorithms and heuristics, what will in our opinion become a necessity in years that come.

References

[Bie08] A. Biere. PicoSAT Essentials. *JSAT* 4, pp. 75–97, 2008.

[Coo71] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *3rd STOC*, pp. 151–158, New York, 1971.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM* 5(7), pp. 394–397, 1962.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM* 7(3), pp. 201–215, 1960.

[ES04] N. Een and N. Sorensson. An Extensible SAT Solver. In *SAT '03*, LNCS 2919, pp. 502–518, S. Margherita Ligure, 2003.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A Rast and Robust SAT Solver, In *DATE'02*, pp. 142–149, Paris, 2002.

[GSK98] C. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search through Randomization. In *15th AAAI*, pp. 431–437, Madison, 1998.

[HHH+06] F. Hutter, Y. Hamadi, H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP'06*, pp. 213–228, 2006.

[Hua07] J. Huang. A Case for Simple SAT Solvers. In *CP '07*, LNCS 4741, pp. 839–846, Providence, 2007.

[KG07] S. Krstić and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *FroCos '07*, LNCS 4720, pp. 1–27, Liverpool, 2007.

[Mar09] F. Marić. Formalization and implementation of modern sat solvers. *JAR*, accepted for publication, 2009.

[NMJ09] M. Nikolić, F. Marić, P. Janičić. Instance-Based Selection of Policies for SAT Solvers. Manuscript submitted.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. of the ACM* 53(6), pp. 937–977, 2006.

[PD07] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *SAT '07*, LNCS 4501, pp. 294–299, Lisbon, 2007.

[XHH+08] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-Based Algorithm Selection for SAT. *JAIR* 32, pp. 565–606, 2008.