06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:19 AM Page 50

# Framework for Generating Configurable SAT Solvers

Bernardo C. Vieira[1], Fabrício V. Andrade[2], and Antônio O. Fernandes[3]

[1,3] C.S. Department, Universidade Federal de Minas Gerais, Antonio Carlos, 6627, Minas Gerais, Brazil
[2] C. S. Department, Centro Federal de Educação Tecnológica de Minas Gerais, Amazonas, 5253, Minas Gerais, Brazil
e-mail: {bcvieira, otavio}@dcc.ufmg.br, vivas@decom.cefetmg.br

**ABSTRACT**

The state-of-the-art SAT solvers usually share the same core techniques, for instance: the watched literals structure, conflict clause recording and non-chronological backtracking. Nevertheless, they might differ in the elimination of learnt clauses, as well as in the decision heuristic. This article presents a framework for generating configurable SAT solvers. The proposed framework is composed of the following components: a Base SAT Solver, a Perl Preprocessor, XML files (Solver Description and Heuristics Description files) to describe each heuristic as well as the set of heuristics that the generated solver uses. This solvers may use several techniques and heuristics such as those implemented in BerkMin, and in Equivalence Checking of Dissimilar Circuits, and also in Minisat. In order to demonstrate the effectiveness of the proposed framework, this article also presents three distinct SAT solver instances generated by the framework to address a complex and challenging industry problem: the Combinational Equivalence Checking problem (CEC). The first instance is a SAT solver that uses BerkMin and Dissimilar Circuits core techniques except the learnt clause elimination heuristic that has been adapted from Minisat; the second is another solver that combines BerkMin and Minisat decision heuristics at run-time; and the third is yet another SAT solver that changes the database reducing heuristic at run-time. The experiments demonstrate that the first SAT solver generated is a faster solver than state-of-the-art SAT solver BerkMin for several instances as well as for Minisat in almost every instance.

**Index Terms:** SAT Solvers, Combinational Equivalence Checking, and Formal Verification.

## 1. INTRODUCTION

Satisfiability (SAT), a well known NP-Complete problem, stands for proving that a propositional logic formula evaluates to true by some variable assignment, or that there is no such assignment. Despite SAT complexity, SAT solving techniques (e.g. SAT solvers) have been successfully used, for almost two decades, on important problems of the semiconductor industry such as Combinational Equivalence Checking (CEC) and Automatic Test Pattern Generation (ATPG). Whereas CEC techniques prove that two combinational circuits are (or not) functional equivalents; ATPG determines inputs that, when applied to the circuit, show a stuck-at-value error. Since SAT solvers have long been used for circuit verification, instances of this type of problem have also been used as benchmarks for new SAT solvers. Competitions as the one arranged by satcompetition.org provide industrial benchmark instances, some of which are derived from CEC problems.

Even though several advances have been made in SAT solving techniques in the last decade, some classes of circuits such as the arithmetic ones continue to be a great challenge for Electronic Design Automation tools. For instance, proving functional equivalence among multipliers and dividers usually takes two or three orders of magnitude upward in time in comparison to other combinational circuits, such as adders, even with the same number of logic gates; specially when SAT solvers are used. Andrade et al [1] presents a comparison among the state-of-the-art SAT solvers by using a benchmark with several CEC instances derived from their circuit generation tool. As demonstrated in that paper, multipliers and dividers continue to challenge SAT solvers performance.

Another conclusion stated in Andrade's paper is that BerkMin [5] was the most suitable SAT solver for the selected benchmark providing the best solving times. Furthermore, proving functional equivalence among multipliers of different architectures, which

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:19 AM Page 51



present few internal structural similarities (ie. Dadda Tree x Wallace Tree), highly increases the complexity. Although BerkMin presented the best results in the referred paper, it is quite hard to theoretically demonstrate the reason why it achieved such good results since each SAT solver is usually implemented with several slightly different heuristics and data structures. Finally, BerkMin source code is not freely available and many of its own implementation details may have not yet been published.

In order to provide a clear opportunity for new advances in SAT solving techniques through tackling the previous mentioned problems, this paper proposes, implements and provides a framework for generating configurable SAT solvers so that the community would be able to freely modify and test new techniques and heuristics. Using the proposed framework, another key contribution is achieved. By means of a simple selection of different previously existent methods in a modular fashion, it is possible to achieve better solving times than those obtained by important and efficient SAT solvers, such as BerkMin, for several instances.

This article has the following outline: Section 2 explains the related work. Section 3 describes the methodology and the implementation details used in the framework for generating configurable SAT solvers. Section 4 discloses the achievements and compares results of generated solvers (SAT solvers instances) with other state-of-the-art SAT solvers. Section 5 exposes the final considerations and possible future works.

## 2. RELATED WORK

Nowadays, the majority of the SAT solvers (DPLL [20] SAT solvers) share the same core which is composed by several techniques that were previously proved successful[1]. The improvements achieved by those techniques may be seen in the following papers: GRASP [12], SATO [17], Chaff [11], zChaff [18].

The most important techniques and heuristics used in currently available SAT solvers are listed in order: clause learning, Unique Implication Points (UIPs) and non-chronological backtracking (used in GRASP); the "Two-Watched Literals" structure (proposed by SATO, and modified by Chaff in the elimination of head and tail constraints); Variable State Independent Decaying Sum (VSIDS) decision heuristic (proposed by Chaff); the First-UIP in the zChaff solver. The "Two-Watched Literals" structure is widely used because it does not require updates when a conflict occurs[2]. In the conflict implication graph, a UIP is a node at the current decision level such that every path from the decision variable to the conflict must go through it. The conflict clause is built by cuts at UIPs. The First-UIP is the partition at the graph that is closest to the conflict[3].

Apart from the similarities, many other heuristics have been proposed by the solvers. The BerkMin solver proposes several changes in the basics structures. Based on observations, the authors organized the set of learnt clauses as a stack. The closest topmost non-satisfied clause was called the current clause. The most active variable from the current clause was chosen as the next variable of the search process. The "activeness" of a variable is defined by the VIDS heuristic, which keeps a counter to each variable, counting the number of times the literals of a variable appears at any clause. Those counters are periodically multiplied by a decay factor, thus guiding the search to the variables of the most recently added clauses. Chaff states that this is a quasi-static search, because it does not depends on the value of the variable and gradually changes as conflict clauses are generated. In BerkMin, the decay factor for the variables was modified to 4, increasing the sensibility to the new added clauses. Conversely, Minisat [2] proposes an aggressive learnt clause elimination procedure that just leaves high activity or binary clauses.

Besides internal data structures and heuristics, another key contribution is the extensibility and modularity of ths SAT solvers. Minisat, for instance, proposed modularity through inheritance. Nevertheless, this article proposes a different kind of modularity, in which the specific data structures and operations of each heuristics and techniques are modularized in separated XML descriptions.

SATenstein [9] proposed a different type of modularization. Starting from the observation that "designing algorithms for hard problems is difficult and time consuming", the authors proposed a generalized and high parameterized solver framework, that includes components from, or based at, other solvers. The parameters control the instantiation and the behavior of those components. This idea is very similar to the proposed approach. The authors, then, use a black-box algorithm (see [7]) to find parameters that represent high-performance instantiations of SATenstein, according to the input SAT data sets. SATenstein is based on SPEAR and PARAMILS [7], but applied to stochastical local search. The major limitation of SATenstein is how to tune its solver since the algorithm to find the parameters is, as stated previously, a black-box one.

[1] A survey on the techniques and modules implemented in state-of-the-art DPLL SAT solvers was presented by Kautz and Selman[18].

[2] More about this data structure is found at Chaff[11].

[3] Explanations at zChaff[18]

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:19 AM Page 52



SATzilla [16] is a "per-instance solver portfolio" implements a new type of modularity. It constructs a model using a predefined set of complete solvers (ie: Minisat, BerkMin). The model is trained based on the run-time of each solver for each SAT data set. On-line, the model, uses the SAT instance features to choose the best solver from the portfolio, and solves the instance. The present approach can be integrated to SATzilla as one can generate solvers that may be added to the portfolio.

Although several SAT solvers make use of modularity concepts, this article proposes and implements a great extension to this concept which is a framework for generating configurable SAT solvers.

## 3. FRAMEWORK FOR GENERATING CONFIGURABLE SAT SOLVERS

### *A. Introduction*

SAT solvers are proposed, implemented and submitted to competitions to have their performance measured, both in time and memory. Heuristics and parameters vary among solvers, and, usually, each solver reimplements its core techniques and heuristics (for instance, the Watched Literals structure) from scratch. Therefore it might be difficult to identify the reasons which influenced a solver to achieve certain results in comparison to other solvers. In order to provide clarity about new heuristics and techniques, as well as a common methodology for SAT solver implementation, the present article proposes a framework for generating configurable SAT solvers. The key idea is to demonstrate a simple, though robust, mechanism for storing and combining several techniques and heuristics, to generate a new SAT solver through simple configuration files.

Figure 1 shows the basic structure of current SAT solvers. A description of a modified version of the classic DPLL can be observed at its top.

The most common and significant (modified) DPLL SAT solver heuristics and techniques are among the following:

- ReduceDB - learnt clause elimination, or database reducing method (also referred as Database Management or DBManag). Includes the condition in which this method is called (called ReduceDBCond);
- Decide - choose the next free variable and its assignment;
- Deduce - BCP (Boolean Constraint Propagation);
- Diagnose - conflict clause learning and non-chronological backtracking;
- Restart - restart the search maintaining some solver state. Includes the condition in which this method is called.

SAT solvers replicate this modified DPLL structure. Many new SAT solvers, such as BerkMin, Minisat and zChaff, could be considered specializations of that modified version of the DPLL SAT solver, as illustrated in the figure 1. Whereas new SAT solvers tend to preserve techniques inherited from its classic version, there are those that change and reimplement these techniques. For instance, BerkMin implements a different decision heuristic compared to DPLL, even though it uses a similar Deduce engine.

As illustrated in Figure 1, the proposed framework generates SAT solvers that may be described as an inheritance of different SAT solvers. The expected advantages of this framework are its robustness and flexibility of new modules implementation. The robustness comes from the fact that the same code is used by several solvers, whereas the flexibility of implementation derives from the modularization of different heuristics and techniques in different files.

### *B. Definition of the Framework Implementation Paradigm*

Since this article proposes a framework for generating SAT solvers which is based on the division of the heuristics and techniques in separate files, one file for each heuristic/technique, the issue we worked on was how to efficiently implement it. To address it, three approaches were considered.

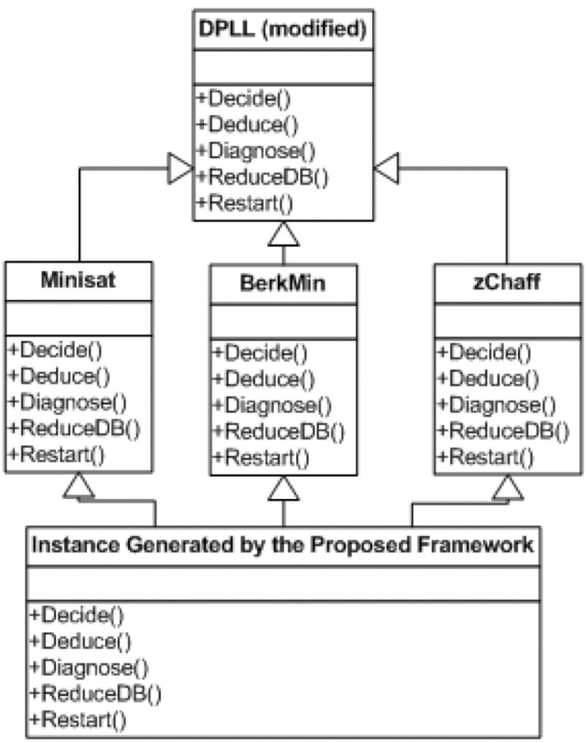


**Figure 1.** Logical view of SAT solvers implementations

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:19 AM Page 53



The first implementation approach consists in concentrate in the Object-Oriented Programming (OOP) paradigm, creating classes of inheritance among the heuristics. This is, basically, the modular approach used by Minisat. For instance, a solver that would implement Berkmin heuristics could be named as *BerkSolver* and it would be child class from the class *Solver*. Besides, for each possible module combination from *BerkSolver* and *Solver* classes, there would be derived classes represented by *Solver1* and *Solver2* as illustrated in Figure 2.

Note that the user should decide which methods to use and which to reimplement according to the base class. Although this approach is supported by the OOP paradigm, it could lead to drawbacks. First, the requirement of extra source codes and extra files to provide the integration among the several inherited classes from different heuristics. Another point to consider is the amount of generated files for each possible combination of heuristics.

Other implementation approach is the use of inheritance class and compilation flags. Thus, instead of an inheritance from different classes, the only inheritance allowed would be from a base skeleton class. Consequently, the desired class would have the behavior of the skeleton class with the changes determined by the compilation flags. For instance, a *#ifdef* and *#endif* directives could be used to select a different type of Decide heuristic. This approach could prevent code replication, since the user should select, according to the chosen flags, which heuristics to instantiate. Finally, a new SAT solver with different heuristics could be created through the compilation of the class inherited from the base skeleton class and the flags chosen.

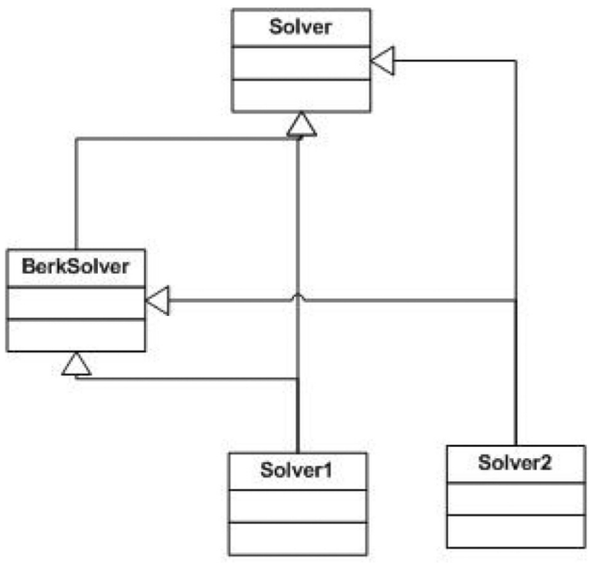


**Figure 2.** Inheritance suggested by the first implementation approach

This approach presents improvements with respect to the first approach; however, it would be difficult to figure the source code of the entire project. This happens because the user does not have a global perspective on how the selected flags alter or implement each part of the solver source code. In addition, the C programming language does not support chained macros. Therefore, this approach requires solid programming skills to enable the creation of a SAT solver which possesses a clear source code.

The last implementation approach considered, takes in account the Aspect Orientation Programming (AOP) paradigm. As a result, snippets of the source code would be included according to a pre-defined event, such as a method call. Note that the main benefits of AOP from the implementation of orthogonal components to the source code are: non functional requirements or logging, as well as less generation of code. The proposed framework was created by implementing the third approach through use of configuration files, C/C++ source codes, and a Perl pre-processor.

### *C. Framework Components and Basic Operation*

The proposed framework is composed of the following components: a *Base Solver,* a *Perl Preprocessor, XML files (Solver Description* and *Heuristics Description* files*)* to describe the heuristics as well as the set of heuristics that the generated solver uses. The framework basic operation is described in Figure 3.

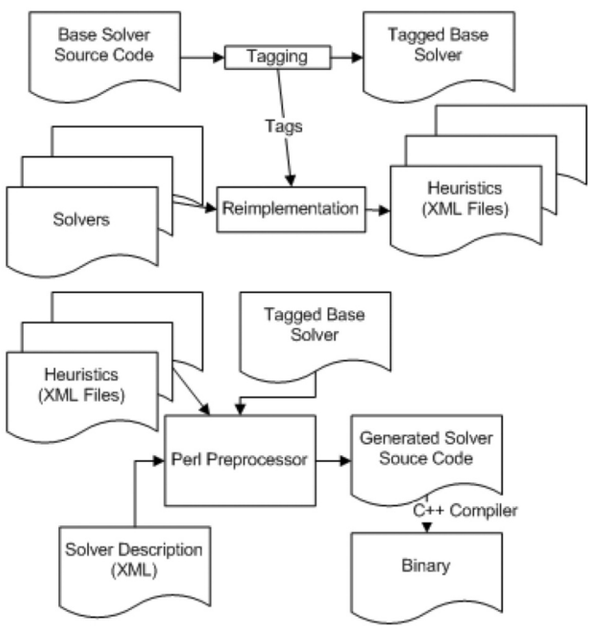


**Figure 3.** Framework for generating configurable SAT solvers

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:19 AM Page 54



First of all, a *Base Solver* is selected for the basic operation of the framework. In the present work, Minisat SAT solver has been selected for its competitiveness, and because its source code is widely available. Then, it is necessary to identify and tag the source code of the *Base Solver*, specially the parts of the code that contain heuristics implementations, since these parts may be replaced by source code of different heuristics. For instance, variable database management or decide procedures from other SAT solvers. In other words, the *Base Solver* Deduce procedure is tagged because this part of the source code may be replaced by another solver (Berkmin or Rsat) Deduce procedure that is stored in XML files.

As shown in Figure 3, heuristics and techniques (algorithms and data-structures) that might be new or taken from other SAT solvers are selected, reimplemented, tagged and stored in XML files. Each XML file also includes the same tags of the *Base Solver* in order to create a unique correspondence among them.

After that, XML file that contains the *Solver Description*, which has a list of modules representing the set of heuristics, techniques, and parameters that the generated solver uses, is created. Thereafter, it is necessary to run a *Perl Preprocessor,* which consults the selected heuristics from the *Solver Description* XML file, and replaces the heuristics source code from the tagged *Base Solver* to the *Heuristics* described in the XML files (*Heuristics Description* files). Note that using a preprocessor allows any code to be produced in replacement of the tags, in any hierarchy and order, which is more maintainable and reusable than storing the source code and object files. This also links the source code and object files, dynamically, as required.

After executing the preprocessor, the output is a source code of a new SAT solver according to user's specification (*Solver Description* file). Then, according to Figure 3, it is necessary to submit the source code to a C++ compiler that, by its turn, outputs the binary file for the new SAT solver.

To provide more details about the implementation architecture, Listings 1 exemplify how a class solver could be tagged. In this example, two types of tags are used; the first indicates where the variable declaration must be inserted, whereas the second illustrates where the actual source code implementation must be put.

Listings 2 describes an *Heuristic Description* file. This file type follows a pattern in which its first line has the **id** that defines the name of the heuristic, in this case, **dbManag**. Each heuristic described has one or more mapping to the source code. Each mapping starts with the tag **macro** and has an attribute named **id** that identifies the marker; other attribute named **type** which defines whether the source code marked inside the tags **<code>** and **</code>** should be appended or overwritten into the existent source code; and yet another attribute named **priority** which defines what type of macro is going to have priority in case of a collision. The overwritten attribute has greater priority with respect to the appended attribute. There are two forms of the **<code>** and **</code>** tags: a source code could be typed inside the tags, or a reference to file could be applied as shown in Listings 3. Finally, Listings 4 describes the *Solver Description* file, which consists of a reference to several heuristics files, since a solver might be seen as a set of heuristics.

```
class USolver: public Solver{
//<!—USolver.h/declarations—>
};
//<!—USolver.h/implementations—>
```

**Listings 1.** Examples of marks or tags used in the Base Solver source code

```
<heuristic id="dbManag">

<macro id="USolver.h/declarations"
type="append" priority="0">
<code>
  protected:
     void minisat_reduceDB( ) ;
     void claDecayActivity( ) ;
     void claBumpActivity( Clause& c ) ;
</code>
</macro>

<macro id="USolver.h/implementations"type="append"
priority="0">
<file name="minisat/dbmanag.implementation.h"/>
</macro>

</heuristic>
```

**Listings 2.** Example of Heuristics Description File

```
<heuristic id="decide_berkmin_minisat">
   <subheuristic id="minisat/decide.xml"/>
   <subheuristic id="berkmin/decide.xml"/>
</heuristic >
```

**Listings 3.** Example of how subheuristics may be used inside a heuristic file

```
<solver id="minisat">
<output folder="src-minisat"/>
    <heuristic file="minisat/dbmanag.xml" />
    <heuristic file="minisat/decide.xml"/>
    <heuristic file="minisat/reducedb.xml"/>
    <heuristic file="minisat/restart.xml"/>
    <heuristic file="minisat/simplify.xml"/>
    <heuristic file="minisat/var.xml"/>
</solver>
```

**Listings 4.** Example of XML Solver Description fle

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:19 AM Page 55



### *D. Base Solver Internal Algorithms and Data Structures*

As mentioned previously, the proposed framework is based on the Minisat implementation. However, several changes in the internal algorithms and data structures were performed in order to fit this base solver into the framework. The most important changes are described in the following paragraphs.

The possibility of run-time changing of heuristics lead to the replication of certain data structures, whereas others were preserved. Both, Minisat and BerkMin, use the "Two-Watched Literals" data structure. They also use a conflict clause learning scheme that applies reverse BCP (resolution) up to the point when there is only one literal at the level of the conflict on the reason side of the implication graph. Those methods were left untouched in the new solver. Moreover, the clause database is shared, even if the heuristic is modified in run-time, as this is essential to a real combination of methods, since they share the same search-space.

The "var activity bumping" was not completely kept, as BerkMin bumps the activity in any intermediate reverse BCP resolution step. This lead to a replication of data structures (array of type double) for the variable activity. The clause activity must also be replicated. There are two clause activities, one for BerkMin, based modules, and one for Minisat, also based modules, because using BerkMin there is no decay factor for clause activity and thresholds.

The major issue with the framework implementation is to define what should change on the module when different solver heuristics are used. For instance, using two identical signature modules, previously allowed to change at run-time, will create one variable to determine which module is called. In the framework present configuration, the user can choose between BerkMin and Minisat database management heuristics, or their changing in run-time (files *minisat/dbmanag.xml*, *berkmin/dbmanag.xml* and *combination/dbmanag.xml*). This is also true for the decision variable (files *minisat/decide.xml*, *berkmin/decide.xml* and *combination/decide.xml*).

Experiments confirm that the best results were achieved when the decay factor was approximately 1 (1/0.995). The file that configures this parameter is *minisat/newparms.xml*. The run time modifications happen at every 50 restarts. BerkMin heuristics were implemented as described by the papers [5] and [4].

### *E. Implemented CEC Heuristics*

CEC allows us to demonstrate the importance of the approach presented previously; therefore, this subsection will provide the CEC implemented modules. Goldberg and Novikov [5], [4] presented modifications that achieved satisfactory results to SAT solving CEC problems. BerkMin[5] implemented:

- Learnt clause database as a stack. The topmost clause is the last learnt clause;
- Decision Heuristic: choice of the variable. When there are unsatisfied conflict clauses, the most active variable clause of the topmost clause of the database stack is chosen. Otherwise the most active free variable of the original clauses is chosen;
- Decision Heuristic: variable sign. When there are unsatisfied conflict clauses, the variable's literal with greater literal activity is chosen, otherwise the literal with greatest nb_two is chosen;
- the new activity counter places (at each clause used in reverse BCP);
- BerkMin's learnt clause elimination heuristic.

Equivalence checking of dissimilar circuits [4] explains the Common Specification (CS) as a mechanism to generalize the notion of structural similarity. In the Common Specification procedure, the structure of the miter circuit, which is a Boolean function, is represented as directed acyclic graph. The two circuits can have intersections between their subgraphs, which are determined by correlation functions. The intersections are equivalences themselves. Then, the specification of the circuit is used to minimize the variable and clause search space, which are used to guide the search at the SAT level. A monotonically increasing function is applied to the variable counters. This function also depends on the variables subgraph depth from the inputs. Goldberg and Novikov adapted the patterns shown by CEC with CS in BerkMin solver, and probably used the adaptations as its strategy 1. They mimic short resolution proofs modifying the decision and restart procedures. The decision procedure provided in the proposed framework may be configured using the file *dcircuit/var.xml*, that chooses from the top most clause the one with greater activity even when the clause is satisfied. The second change at the decision procedure is the selection of the variable with greater activity multiplied by a monotonically growing function of the level of that variable, where level is the topological level of the variable starting from the inputs of a circuit.

In the restart procedure the change is the implementation of heavy and light restarts (configured by *dcircuit/restart.xml* file). Light restarts occurs at depth greater than 15 at the second conflict found at the current search tree. Heavy restarts occurs at each 500 conflicts. In heavy restarts, the database is cleaned, whereas in light it is not.

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:19 AM Page 56



### *F. Using and Validating the Proposed Framework*

As previously explained, the proposed framework that allows users to choose, implement and combine heuristics and techniques according to their needs. In order to demonstrate its effectiveness, this article presents three distinct SAT solver instances, generated by the framework, to be used in a complex and important industry problem: the SAT-based Combinational Equivalence Checking problem. The first instance consists of adapting the aggressive database reducing heuristics used in Minisat to BerkMin. Minisat's database reducing heuristics is more aggressive because it keeps just binary clauses or those with activity greater than a factor of the current activity decay value and the number of clauses. The second instance consists of changing the decision heuristic at run-time. The third instance consists of changing the database reducing heuristic at run-time. Considering the first instance, it is arguable that smaller clauses can improve the run-time since less cache misses are generated; and according to Malik et al[11], 90% of the time is spent at Boolean Constraint Propagation (BCP). Regarding the second and third instances, a change in run-time in the decision heuristic, or in the database reducing method, allows the search for different solution spaces, which is also achieved by the use of restarts and clause recording. Dynamic changing of decision heuristics is presented at [3]. Restarts and clause recording, according to Gomes et al [6] brought SAT solvers in the direction of general resolution, which is considered the main factor of efficiency of SAT solvers in real world problems. Therefore, the last two instances generated by the framework will test the power of changing heuristics at run-time.

## 4. RESULTS

This section presents the results achieved by applying CEC instances to the proposed SAT solver, with different combinations of heuristics, generated by the framework. The results are compared to Minisat 2.0 [2], RSAT [10], Siege v4 [14] seed 100 and BerkMin561 strategy 1 [5]. The main objective of the results is to compare BerkMin and Minisat to the generated solvers, since they are combinations of those solvers.

The benchmark is composed of miters of adders, multipliers and dividers, from size 4 to 32, generated using BenCGen [1]. This type of benchmark is used instead of the ISCAS 85 benchmarks as the latter do not represent a challenge to the recent state-of-the-art formal verification tools. Other benchmarks, such as ISCAS 89 and IWLS-2005, were not used because they are either outdated due to Intellectual Properties, or have a small number of combinational circuits. Conversely, BenCGen generates parametrized descriptions of $n \times n$ multipliers, dividers and adders. After generating the circuits with BenCGen, the miter is created using a sintetized (using the ABC tool [13]) and non sintetized version of the circuit. Due to space restrictions, the complete results are available at *http://sites.google.com/site/modularsat/*. Experiments were also made with part of the DIMACS benchmark for satisfiability. The results were obtained on an Athlon IV 3.0 GHz with 2 GB of memory running Fedora Core Linux OS 10.0. The total verification time was measured in seconds. Four instances of the modular solver were proposed. The first instance (Instance 1, or I1) uses the implemented BerkMin and Dissimilar Circuits modules, except the database reducing module, that is taken from Minisat. The second instance (Instance 2, or I2) is a modification of Instance 1 and changes the decision heuristic at run-time[4], between BerkMin and Minisat. The third instance (Instance 3) is another modification of Instance 1, but changes the database reducing module at run-time. The last instance, BerkMin*, is the implementation of BerkMin and Dissimilar Circuits, and represents the experimental control.

Table I presents the results of the proposed solvers, Minisat, BerkMin, Siege and RSAT for the Wallace Tree Multiplier (WTM). As the circuit size grows, the Instance 1 solver provides much better results, as it starts to dramatically reduce the solving time. Moreover, Instance 1 solves many more instances than the other solvers assuming the 10,000 second time threshold.

Instance 1 achieved better results in 22 out of 26 instances when compared to the other solvers.

Tables II and III present the results, of the same set of solvers, for of the proposed solvers, Minisat, BerkMin, Siege and RSAT for the Carry Look Ahead Multiplier (CLAM) and for the Non-Restoring Divider (NRD). For the Non-Restoring Divider and the Carry Look Ahead Multiplier, Instance 1 performs better mainly between 9 and 12 bits. For those circuits, the Instance 1 achieved better results in 11 out of 30 instances (BerkMin* and Instance 3 won on some of them). In the other cases, generally BerkMin is faster. The possible explanation is that BerkMin uses a monotonically increasing function to sort the variable's activity which the framework does not yet implement.

While Instance 1 achieved consistent results for WTM, the other two instances did not. However, they could be of use as heuristics for other types of SAT

[4] Again, using its own XML description.

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:20 AM Page 57



**Table I.** CEC Results of two Wallce Tree Multiplier copies

| Results of Combinational Equivalence Checking for two Redundant Copies of a Wallace Tree Multiplier | | | | | | | | | | I-1Speed-up | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **# Clauses** | **Rsat** | **Minisat** | **Siege** | **Berkmin** | **I-1** | **I-2** | **B*** | **I-3** | **Berkmin** | **Minisat** |
| 07x07 | 4392 | 7.071 | 3.460 | 6.566 | 3.370 | 3.319 | 4.343 | 3.398 | 4.494 | 1.5 | 4.2 |
| 08x08 | 5968 | 71.762 | 30.592 | 64.466 | 55.560 | 6.481 | 8.687 | 53.539 | 49.438 | 757 | 372 |
| 09x09 | 7882 | 656.789 | 287.092 | 1487.460 | 220.150 | 13.337 | 20.139 | 482.487 | 682.676 | 1550 | 2052 |
| 10x10 | 9980 | 4742.416 | 1585.400 | >10000 | 265.480 | 24.702 | 34.360 | 459.993 | 1158.870 | 974 | 6318 |
| 11x11 | 12422 | >10000 | >10000 | - | 320.360 | 45.849 | 109.766 | 692.031 | 418.234 | 598 | - |
| 12x12 | 15254 | - | - | - | 494.920 | 45.300 | 224.696 | 659.738 | 655.669 | 992 | - |
| 13x13 | 18390 | - | - | - | 960.390 | 95.361 | 315.444 | 662.513 | 1785.760 | 907 | - |
| 14x14 | 21808 | - | - | - | 1131.380 | 198.663 | 1544.030 | 1349.530 | 2281.180 | 469 | - |
| 15x15 | 27102 | - | - | - | 1601.910 | 256.385 | 2087.330 | 556.870 | 1552.340 | 524 | - |
| 16x16 | 30084 | - | - | - | 2213.730 | 345.930 | 2784.490 | 1032.620 | 2974.510 | 539 | - |
| 17x17 | 34732 | - | - | - | 2762.060 | 427.325 | >10000 | 1277.020 | 3241.660 | 546 | - |
| 18x18 | 41576 | - | - | - | 3006.490 | 598.568 | - | 1833.400 | 4736.130 | 402 | - |
| 19x19 | 47272 | - | - | - | 3292.240 | 912.636 | - | 2644.370 | 9565.280 | 260 | - |
| 20x20 | 51342 | - | - | - | 5080.990 | 1233.600 | - | 3888.640 | >10000 | 311 | - |
| 21x21 | 60062 | - | - | - | 5465.910 | 1277.150 | - | 4527.720 | - | 327 | - |
| 22x22 | 67218 | - | - | - | 6094.130 | 3912.350 | - | 4964.160 | - | 55 | - |
| 23x23 | 72188 | - | - | - | 7438.480 | 2678.310 | - | 6538.850 | - | 177 | - |
| 24x24 | 83200 | - | - | - | 6999.700 | 2144.140 | - | 8374.540 | - | 226 | - |
| 25x25 | 88810 | - | - | - | >10000 | 3518.170 | - | >10000 | - | - | - |
| 26x26 | 101126 | - | - | - | - | 4311.210 | - | - | - | - | - |
| 27x27 | 111040 | - | - | - | - | 7892.150 | - | - | - | - | - |
| 28x28 | 121400 | - | - | - | - | 7234.220 | - | - | - | - | - |

**Table II.** CEC Results of two Carry Look Ahead Multiplier copies

| Results of Combinational Equivalence Checking for two Redundant Copies of a Carry Look Ahead Mutiplier | | | | | | | | | | I- Speed-up | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **# Clauses** | **Rsat** | **Minisat** | **Siege** | **Berkmin** | **I-1** | **I-2** | **B*** | **I-3** | **Berkmin** | **Minisat** |
| 07x07 | 3158 | 2.433 | 2.166 | 6.025 | 3.690 | 8.052 | 9.897 | 3.284 | 4.987 | -54 | -73 |
| 08x08 | 4646 | 18.315 | 22.946 | 98.002 | 32.420 | 40.802 | 45.743 | 42.512 | 30.675 | -20 | -43 |
| 09x09 | 6586 | 49.530 | 91.338 | 1075.730 | 147.640 | 104.055 | 111.396 | 203.436 | 284.996 | 41 | -12 |
| 10x10 | 9062 | 952.256 | 485.919 | 3416.990 | 386.060 | 216.800 | 666.551 | 868.389 | 1909.470 | 78 | 124 |
| 11x11 | 12166 | 2689.807 | 8804.550 | 4680.320 | 586.510 | 372.308 | 5029.800 | 2505.130 | 4012.240 | 57 | 2264 |
| 12x12 | 15998 | >10000 | >10000 | >10000 | 829.240 | 838.303 | >10000 | 9147.690 | >10000 | -1.0 | - |
| 13x13 | 20666 | - | - | - | 1259.000 | 1553.610 | - | >10000 | - | -18 | - |
| 14x14 | 26286 | - | - | - | 2030.420 | 2505.960 | - | - | - | -18 | - |
| 15x15 | 32982 | - | - | - | 2458.150 | 4639.290 | - | - | - | -47 | - |
| 16x16 | 40886 | - | - | - | 3494.290 | 8930.680 | - | - | - | -60 | - |
| 17x17 | 50138 | - | - | - | 5203.190 | >10000 | - | - | - | - | - |
| 18x18 | 60886 | - | - | - | 6880.340 | - | - | - | - | - | - |
| 19x19 | 73286 | - | - | - | 9482.060 | - | - | - | - | - | - |

**Table III.** CEC Results of two Non-Restoring Divider copies

| Results of Combinational Equivalence Checking for two Redundant Copies of a Nor Restoring Divider | | | | | | | | | | I- Speed-up | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **# Clauses** | **Rsat** | **Minisat** | **Siege** | **Berkmin** | **I-1** | **I-2** | **B*** | **I-3** | **Berkmin** | **Minisat** |
| 07 | 2198 | 37.990 | 16.385 | 33.425 | 19.030 | 3.317 | 11.543 | 10.130 | 13.776 | 473 | 394 |
| 08 | 2847 | 281.643 | 220.935 | 322.359 | 40.430 | 5.389 | 33.600 | 16.313 | 21.582 | 650 | 3999 |
| 09 | 3580 | 2326.343 | 1579.840 | 2067.720 | 41.010 | 21.908 | 100.361 | 20.704 | 59.973 | 87 | 7111 |
| 10 | 4397 | >10000 | >10000 | 6268.130 | 64.860 | 43.311 | 627.817 | 41.970 | 126.256 | 49 | - |
| 11 | 5298 | - | - | >10000 | 90.970 | 70.905 | 2354.630 | 70.095 | 290.436 | 28 | - |
| 12 | 6283 | - | - | - | 120.510 | 166.049 | 4859.400 | 220.908 | 489.522 | -27 | - |
| 13 | 7352 | - | - | - | 262.010 | 169.038 | >10000 | 429.191 | 1149.660 | 55 | - |
| 14 | 8505 | - | - | - | 217.230 | 410.910 | - | 600.733 | 1824.100 | -47 | - |
| 15 | 9742 | - | - | - | 380.680 | 745.421 | - | 1055.040 | 3015.700 | -48 | - |
| 16 | 11063 | - | - | - | 504.790 | 3662.640 | - | 1538.610 | 7805.340 | -86 | - |
| 17 | 12468 | - | - | - | 680.900 | 1997.220 | - | 3162.390 | 9778.570 | -65 | - |
| 18 | 13957 | - | - | - | 898.510 | 2942.230 | - | 4449.380 | 7318.760 | -69 | - |
| 19 | 15530 | - | - | - | 1379.190 | 6071.070 | - | 6775.960 | >10000 | -77 | - |
| 20 | 17187 | - | - | - | 1636.410 | >10000 | - | >10000 | - | - | - |
| 21 | 18928 | - | - | - | 2171.930 | - | - | - | - | - | - |
| 22 | 20753 | - | - | - | 2935.480 | - | - | - | - | - | - |
| 23 | 22662 | - | - | - | 4226.850 | - | - | - | - | - | - |
| 24 | 24655 | - | - | - | 5086.610 | - | - | - | - | - | - |
| 25 | 26732 | - | - | - | 5953.820 | - | - | - | - | - | - |
| 26 | 28893 | - | - | - | 8800.340 | - | - | - | - | - | - |

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:20 AM Page 58



problems.

Concluding, the generated solvers are most beneficial in situations where different heuristics are better to be applied at different types of circuits, as the same code is used in any situation. This is believed to make the code more robust, as the code is tested in many occasions.

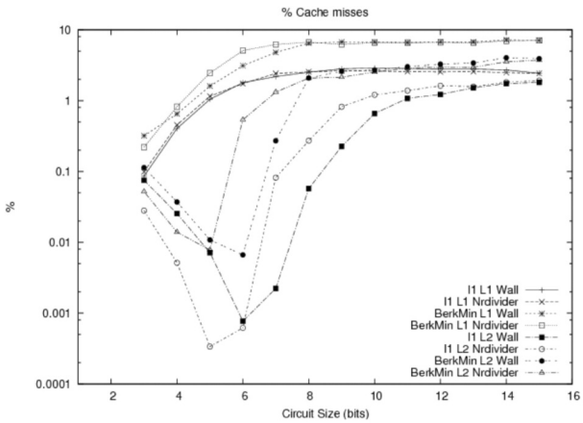


**Figure 4.** Cache profiling graph for WTM and Nrdividers (circuit size x percent cache miss)

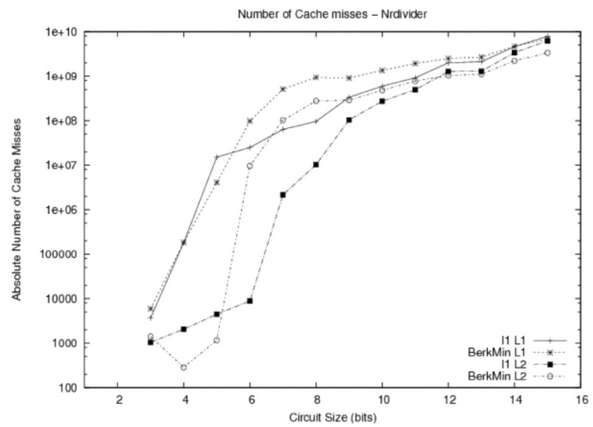


**Figure 5.** Cache profiling graph for Non-Restoring Dividers (circuit size x number of cache misses)

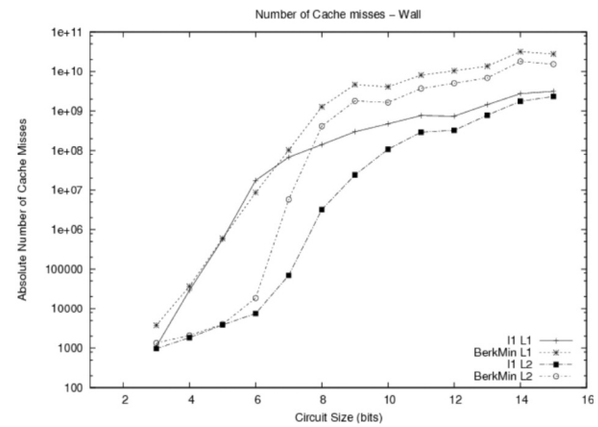


**Figure 6.** Cache profiling graph for Wallace-Tree Multiplier (circuit size x number of cache misses)

Since a lower cache miss rate was predicted to occur in Instance 1 Solver, a cache profiling using Valgrind [15] was made to the Non Restoring Divider and the Wallace Tree Multiplier. The results are found at Figure 4, 5 and 6 for data access, Level 1 (L1) and Level 2 (L2) caches. The organization of the cache is L1 2-way associative, 64 KB,

L2 8-way associative, 521KB; with 64B block size. Although BerkMin has a higher percentage of cache misses in both circuits, it has a lower absolute number of cache misses for the divider. In 14x14 bits, the absolute number of Data L1 cache misses is the same for Instance 1 and BerkMin. However, the number of BerkMin's Data L2 cache misses is lower than Instance 1's. Therefore, the verification time for BerkMin is also lower. For the 15x15 bits, the Instance 1's Data L1 cache misses is greater than BerkMin's. For the Wallace Tree, Instance 1 has the lowest absolute number of cache misses, starting from 7x7 bits. The percentage of BerkMin's Data L1 cache misses is always greater than Instance 1. The Instance 1 percentage of cache misses is lower then BerkMin for each of the circuits represented in Figure 4 because the inherited Minisat clause database is composed of smaller clauses, and also because the variable is chosen from the topmost clause. If the clauses, which are ordered, are smaller and the next variable is chosen from the topmost clause, then it is expected that when Instance 1 requests a clause it is already at the cache, due to locality reference. To the nrdivider, the problem is that those clauses are not guiding the search to a promising search space, at least not as good as Berkmin, which is solving the problem in less time (with less number of absolute cache access).

## 5. CONCLUSION AND FUTURE WORK

A framework for generating configurable SAT solvers was proposed and implemented. This framework is composed of the following components: a Base SAT Solver, a Perl Preprocessor, XML files (Solver Description and Heuristics Description) to describe each heuristic as well as the set of heuristics that the generated solver will use

To demonstrate the effectiveness of the proposed approach, three instances were proposed for improving performance for the CEC problem. They were then checked in the new solver. The first instance (using Minisat's database reducing heuristic) looks to be the more promising for the CEC problem. Also, the proposed framework permits code reuse and fast testing.

The analysis of the percentage and the total value of cache misses have shown that Instance 1 have the lowest percentage of cache misses. Therefore, the

06-Bernardo-AF:06-Bernardo-AF 8/19/11 6:20 AM Page 59



smaller learnt clauses really lower the percentage of cache misses. However BerkMin have a lower absolute number of cache misses for the Nrdivider, making it faster than Instance 1. This shows that the decision heuristic, or the BerkMin's maintained learnt clauses, allows visits to more promising search spaces.

As a future work, an structure ordering of the initial assignments should be studied, ie. J-frontier. Moreover the use of the multiplicative function at the sorting of variable's activity should be tested. The restart period should also be increased periodically to guarantee termination as stated by Zhang and Malik [19]. The profiling of cache misses between the different solvers should also be done, to verify the expected decrease provided by the use of the Minisat clause database heuristics. Other cache orientations should also be tested.

The automatic configuration of the parameters that control the instantiation and the behavior of the modules is a big improvement and should be implemented for the presented SAT solver. This would allow a future comparison with SATenstein which already implements these features.

## REFERENCES

[1] F. V. Andrade, L. M. Silva, and A. O. Fernandes, "BenCGen: a digital circuit generation tool for benchmarks", in *SBCCI '08: Proceedings of the 21st Annual Symposium on Integrated Circuits and System Design*, pages 164–169, New York, NY, USA, 2008. ACM

[2] N. Eén and N. Sörensson, "An extensible sat-solver", in E. Giunchiglia and A. Tacchella, editors, SAT, volume 2919 of Lecture Notes in Computer Science, pages 502–518. Springer, 2003.

[3] E. Giunchiglia and A. Tacchella, editors, "Theory and Applications of Satisfiability Testing", 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, volume 2919 of Lecture Notes in Computer Science. Springer, 2004.

[4] E. Goldberg and Y. Novikov, "Equivalence checking of dissimilar circuits.", In IWLS-2003 12th International Workshop on Logic and Synthesis, 2003.

[5] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust satsolver.", Discrete Applied Mathematics, 155(12):1549 – 1561, 2007.

[6] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability Solvers, Handbook of Knowleadge Representation, chapter 2. ELSEVIER, 2008.

[7] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu, "Boosting verification by automatic tuning of decision procedures.", In FMCAD '07: Proceedings of the Formal Methods in Computer Aided Design, pages 27–34, Washington, DC, USA, 2007. IEEE Computer Society.

[8] H. Kautz and B. Selman, "The state of sat. Discrete Applied Mathematics", 155(12):1514 – 1524, 2007. SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[9] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Satenstein: automatically building local search sat solvers from components.", In IJCAI'09: Proceedings of the 21st international jont conference on Artifical intelligence, pages 517–524, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[10] Knot, "Rsat 2.0: Sat solver description", Technical report, Automated Reasoning Group, Computer Science, 2007.

[11] S. Malik, Y. Zhao, C. F. Madigan, L. Zhang, and M. W. Moskewicz, "Chaff: Engineering an efficient sat solver.", Design Automation Conference, 0:530–535, 2001.

[12] J. P. Marques-Silva and K. A. Sakallah, "Grasp: A search algorithm for propositional satisfiability.", IEEE Transactions on Computers, 48(5):506–521, 1999.

[13] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification.", Electrical Engineering and Computer Sciences UC Berkley, http://www.eecs.berkeley.edu/ alanmi/abc/, 2009.

[14] L. Ryan, "Efficient algorithms for clause-learning sat solvers.", Master's thesis, Simon Fraser University, 2002.

[15] Valgrind, "Valgrind 3.5.0.", Valgrind Developers, http://valgrind.org/, 2010.

[16] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat.", J. Artif. Int. Res., 32(1):565–606, 2008.

[17] H. Zhang, "Sato: An efficient propositional prover.", In CADE-14: Proceedings of the 14th International Conference on Automated Deduction, pages 272–275, London, UK, 1997. Springer-Verlag.

[18] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver.", In ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

[19] L. Zhang and S. Malik., "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications.", In DATE '03: Proceedings of the conference on Design, Automation and Test in Europe, page 10880, Washington, DC, USA, 2003. IEEE Computer Society.

[20] Davis, M., Logemann, G. and Loveland, D., "A machine program for theorem-proving", Commun. ACM, 1962, 5(7):394-397.