SAT Race 2015[☆]

Tomáš Balyo[a], Armin Biere[b], Markus Iser[a], Carsten Sinz[a]

[a] *Karlsruhe Institute of Technology (KIT)*
Department for Informatics
Building 50.34, Am Fasanengarten 5, 76131 Karlsruhe, Germany
{tomas.balyo,markus.iser,carsten.sinz}@kit.edu

[b] *Johannes Kepler University Linz (JKU)*
Faculty of Engineering and Natural Sciences
Altenbergerstr. 69, 4040 Linz, Austria
biere@jku.at

Abstract

Boolean satisfiability (SAT) solving is one of the most competitive research areas of theoretical computer science. The performance of state-of-the-art SAT solvers has been continuously improving in the last decades and has reached a level where SAT solvers can be employed to solve real world problems in fields such as hardware and software verification, automated planning and many others. One of the important driving forces of this progress are the yearly organized (since 2002) SAT competitions. In this paper we describe the 2015 SAT Race that featured the traditional sequential and parallel tracks (with 64 core computers) and introduced the Incremental Library Track, which is particularly interesting for developers of SAT based applications. We describe the 2015 SAT Race and provide a detailed analysis of its results.

Keywords: SAT, Competition, SAT-Race

1. Introduction

One of the most studied problems of theoretical computer science is the Boolean satisfiability (SAT) problem. SAT solving has many practical applications, and SAT solvers are used in the background as high performance reasoning engines in several A.I. applications such as automated planning and scheduling [1], formal verification [2] or automated theorem proving [3]. Despite the fact, that SAT is NP-complete [4] the performance of state-of-the-art SAT solvers has increased dramatically in the last decades thanks to the invention of advanced heuristics [5], preprocessing and inprocessing techniques [6] and data structures that allow efficient implementation of search space pruning [5].

[☆] baldur.iti.kit.edu/sat-race-2015

One of the ways to keep up the driving force in improving SAT solvers is the organization of SAT competitions. The first SAT competition was organized in 1992 [7] followed by the second in 1993 [8] and the third in 1996. Since 2002 a SAT competition – sometimes under the names SAT Race and SAT Challenge – is organized every year and is associated with the SAT conference[1].

SAT Races – the first being organized in 2006 – put a strong emphasis on application benchmarks and are generally on a smaller scale (fewer benchmarks, smaller time limits) than SAT Competitions, which also include tracks for random and "crafted" formulas, the latter being purposefully built to challenge current SAT algorithms. A more detailed overview of the history of the SAT competitions can be found in [9].

Even though sharing the same motivation, the SAT Race organized in 2015 was different from previous competitions in several aspects.

- Five runs were performed for each benchmark/solver pair in the Main Track (in previous competitions only one).
- We used a higher number of cores (64) in the Parallel Track than previous competitions (16–32).[2]
- For the first time an Incremental Library Track was organized and a new standard interface for incremental SAT solving (IPASIR) was introduced.

This paper describes the 2015 SAT Race, its organizational details and results. Large part of the paper is devoted to the Incremental Track and the detailed description of the proposed incremental interface – IPASIR. We hope that IPASIR (or its extension) becomes a standard interface for incremental SAT solver implementations.

2. Preliminaries

A *Boolean variable* is a variable with two possible values *True* and *False*. By a *literal* of a Boolean variable x we mean either x or $\overline{x}$ (*positive* or *negative literal*). A *clause* is a disjunction (OR) of literals. A *conjunctive normal form* (CNF) *formula* is a conjunction (AND) of clauses. A clause can be also interpreted as a set of literals and a formula as a set of clauses. A *truth assignment* ϕ of a formula F assigns a truth value to its variables. The assignment ϕ satisfies a positive (negative) literal if it assigns the value True (False) to its variable and ϕ satisfies a clause if it satisfies at least one of its literals. Finally, ϕ satisfies a CNF formula if it satisfies all of its clauses. A formula F is said to be *satisfiable* if there is a truth assignment ϕ that satisfies F. Such an assignment is called a *satisfying assignment*. The *satisfiability problem* (SAT) is to find a satisfying assignment of a given CNF formula or determine that it is unsatisfiable.

[1] Full name: International Conference on Theory and Applications of Satisfiability Testing
[2] However, 32 of the cores are due to hyper-threading.

Most current complete state-of-the-art SAT solvers are based on the conflict-driven clause learning (CDCL) algorithm [10]. For a detailed description of CDCL refer to [11].

3. Competition Overview

The 2015 SAT Race featured three tracks. Two traditional tracks – the Sequential and Parallel Tracks and the Incremental Library Track, which had its debut. An overview of all participants and their solvers is presented in Table 1. Each participant was allowed to submit at most two solvers (or two versions of one solver) to each track. Except for the Incremental Library Track the submission of source code was optional. The following sections describe organizational aspects of the tracks.

4. Main Track

The Main Track is the most popular track of each SAT competition. In total 28 solvers (solver versions) developed by 18 different groups participated in the 2015 SAT Race. Except for CBPeneLoPe each solver had at least one version participating in the Main Track. See Table 1 for a complete list of solvers and their authors.

In the remainder of the section we describe the benchmarks submitted for the competition, the method we used to select the final set of benchmark problems and how the results were evaluated.

4.1. Submitted Benchmarks

In the following text we briefly describe the 6 benchmark families submitted for 2015 SAT Race. All of them are – in the spirit of the SAT Races – considered application or industrial benchmarks. The detailed descriptions of the benchmarks can be found on the website of the SAT Race[3].

Factorization by Joe Bebel. These satisfiable instances encode the factorization of the products of pairs of large prime numbers. The generator is based on Karatsuba multiplication.

Pseudo-Industrial Random by Jesús Giráldez-Cru and Jordi Levy. The generator is using the Community Attachment model to create random instances with high modularity (a property characteristic for "real-world" problems).

Modulo Game Solving by Tobias Sebastian Hahn, Norbert Manthey and Tobias Philipp. The Modulo game is a combinatorial puzzle where tiles are to be placed on a field such that the sum of all overlaying values of a cell sums up to a multiple of a predefined value.

[3] `http://baldur.iti.kit.edu/sat-race-2015/index.php?cat=downloads`

Solvers	Authors	M	P	I
Ratselfax	Jan Bruns	1	–	–
CryptoMiniSat	Mate Soos, Marius Lindauer	2	2	2
multi-SAT	Sajjad Siddiqui, Jinbo Huang	2	–	–
OR-Tools	Frederic Didier	1	–	–
CBPeneLoPe	Tomohiro Sonobe	–	1	–
COMiniSatPS	Chanseok Oh	2	–	2
GlueMiniSat	Hidemoto Nabeshima, Koji Iwanuma, Katsumi Inoue	2	–	–
Riss	Lucas Kahlert, Franziska Krüger Norbert Manthey, Aaron Stephan	2	2	2
Glucose	Gilles Audemard, Laurent Simon	2	2	1
(Para)Glueminisat	Seongsoo Moon, Inaba Mary	1	1	–
abcdSat, MiniSAT_BCD	Jingchao Chen	2	–	–
(Lin|Plin|Treen)geling	Armin Biere	2	2	–
satUZK	Alexander van der Grinten	1	1	1
Glucose Comm. Switch.	Hitoshi Togasaki	1	–	–
Nigma	Chuan Jiang, Gianfranco Ciardo	2	–	–
BreakIDGlucose	Jo Devriendt, Bart Bogaerts	1	–	–
DCCASatToRiss	Chuan Luo, Shaowei Cai, Wei Wu, Kaile Su	1	–	–
CCAglucose2	Shaowei Cai, Chuan Luo, Kaile Su	1	–	–
Glucose nbSat	Chu Min Li, Fan Xiao, Ruchu XU	2	–	–

Table 1: The list of solvers participating in the 2015 Sat Race. The columns labeled M, P and I indicate the number of solvers (solver versions) submitted to the Main, Parallel and Incremental Tracks respectively.

Single Track Gray Codes by Norbert Manthey. These instances allow the construction of (Single Track) Gray codes by searching a solution to a specification.

Multi-Robot Path Planning by Pavel Surynek. The instances model the question whether a set of robots can find paths in a grid with obstacles in a given number of time steps.

Verification of Cryptographic Algorithms by Aaron Tomb. These benchmarks are derived from the problem of comparing reference and production implementations of cryptographic algorithms.

4.2. Benchmark Selection

Since the number of available benchmark problems for SAT solving is huge while computational resources are limited it is necessary to somehow select a subset of the instances to be used for a competition. A good competition benchmark collection should have the following properties.

- The instances are not too easy – Comparing solvers on trivial instances that any solver can solve is not interesting and can be disadvantageous for complex solvers using advanced and novel techniques.
- The instances are not too hard – Instances that none of the participating solvers can solve in the given time limit do not provide any information about the relative performance of the competing solvers.
- The selection is unbiased and fair – A selection can be favoring a certain solver or group of solvers and/or being disadvantageous for others.
- Many new instances are included – Using the same (or very similar) instances every year could result in solvers being over-optimized for competition benchmarks and not performing well on other problems of interest.

We used the following selection algorithm to select 300 instances for the Main (and 100 for the Parallel) Track from among the newly submitted benchmarks and the benchmarks used in the last years (2014) SAT competition. The selection method is similar to the ones used in the previous competitions.

- First we randomly selected 3 out of the top 7 solvers from the previous competition (participants of the Sequential and Application SAT+UNSAT Tracks of the SAT Competition 2014).
- We ran the selected solvers on all the benchmarks (newly submitted and 2014 SAT Competition) with a 1 hour time limit.
- Let us define the hardness of an instance as the total time the three solvers spent on it (maximum is 3 hours in the case that none of the solvers could solve the instance within the time limit). Using the hardness value, we randomly selected 300 instances based on the normal distribution with mean 1.5 and standard deviation 1.

- Those 300 instances were used for the Main Track and the hardest 100 among them for the Parallel Track.

The final selection contained 167 new and 133 old benchmark problems. Therefore the goal of including many new benchmarks was satisfied. As for the fairness, this selection method is not ideal. Its main drawback is that we are using the previous winning solvers to select new benchmarks. Therefore instances that are too hard for them are not selected which can be disadvantageous for other (innovative) solvers that might be able to solve them. On the other hand, benchmarks that are easy for the previous winners but hard for others are also omitted resulting in a disadvantage for the previous winners. Designing a fair benchmark selection method is a complex issue which should be studied further and improved in the following competitions.

4.3. Evaluation and Prizes

In this year's SAT Race we did five runs for each solver benchmark pair[4]. The solvers were compared based on the average number of problem instances solved within the time limit of one hour in the five runs.

The evaluation was performed using the StarExec cross community logic solving service developed at the University of Iowa[5] [12]. StarExec consists of 192 nodes with Intel(R) Xeon(R) CPU E5-2609 processors, having four compute cores (no hyper-threading) running at 2.40GHz (2393 MHZ) and 256GB of main memory using the Red Hat Enterprise Linux[6] operating system. The participants uploaded their solvers in the form of binary executables using a web interface, therefore the compilation was performed by the participants. Our experience with StarExec was very positive, it is a very convenient tool for organizing a SAT (or other) competitions.

Three prizes were given out to the top three solvers (solving the highest number of instances) and a fourth (special) prize to the "most innovative" solver. The special prize was selected by a ranking method that uses the number of solved instances not solved by the top three solvers of the track. Each solver received for each instance it solved:

- 4 points if none of the top three solvers solved the instance.
- 2 points if exactly one of the top three solvers solved the instance.
- 1 point if exactly two of the top three solvers solved the instance.
- 0 points if all the top three solvers solved the instance.

The special prize was awarded to the solver receiving the highest number of points.

[4]Initially we planned only one run, but since the solvers performed very well and the measurements were finished much sooner than expected we were able to execute four extra runs. In hindsight, it might have been better to increase the number of benchmarks instead.

[5]`https://www.starexec.org/starexec/public/about.jsp`

[6]Server version release 7.2

5. Parallel Track

Parallel computing is a hot research topic in computer science, especially since parallel computers became ubiquitous.

However, there are inherent limitations to the efficient parallelizability of resolution-based SAT procedures. This is in part due to the structure of resolution proofs [13], but also due to the fact that most sequential SAT solvers spend 80% of their runtime by doing unit propagation [14] which is a P-complete problem [15]. Still, modern parallel solvers such as ManySat [16] or Plingeling [17] often achieve superlinear speedups for many benchmarks.

5.1. Portfolios

A common way of designing a parallel SAT solver is the so called portfolio approach. A portfolio is a collection of SAT solvers (different SAT solvers or different versions of the same SAT solver). The solvers are ran on the same problem in parallel until one of them finds a solution. The solvers can additionally exchange useful information such as learned clauses or variable activity statistics. All current state-of-the-art parallel SAT solvers are based on this approach (judging by the results of recent SAT competitions).

Nevertheless, in the context of SAT competitions the portfolio approach is somewhat problematic. As demonstrated by the solver PPfolio in the 2011 SAT Competition, it is possible to win several tracks of the competition by just taking the best solvers from the previous competition and trivially combine them using a shell script into a portfolio. The author of PPfolio argues that such a simple portfolio solver can serve as approximating the "virtual best solver" (see Sec. 7). But he also "shamelessly claims" [18] that "it's probably the laziest and most stupid solver ever written" which "does not even parse the CNF" and "knows nothing about the clauses". A portfolio solver winning the competition can be very demotivating for the developers of core solvers since someone else is winning with their solver.[7]

To avoid this situation in the 2012 SAT Challenge and 2013 SAT Competition a special track was created for portfolio solvers, which were not allowed to enter any of the other tracks. In the 2014 Sat Competition portfolio solvers were not allowed to participate in any of the tracks. In our competition we took a different approach. Portfolio SAT solvers were allowed to participate in each track provided that the authors of the portfolio had consulted all the authors of the used core solvers and received permission for such usage.

5.2. Benchmarks and Evaluation

The hardest 100 instances of the 300 instances selected for the Main Track (see Section 4.2) were used as benchmarks for the Parallel Track. The hardness

[7] It should be noted that non-portfolio solvers are often derived from existing solvers, too. However, they typically give reference to the original solver, e.g., by having a name derived from the original solver's name. Moreover, some competitions included a "Hack Track", in which small modifications to an existing solver can be submitted.

is defined in Section 4.2. Eleven solvers (solver versions) participated in the Parallel Track. The list of participating solvers is displayed in Table 1.

Due to limited computation resources only one evaluation run was performed for each solver-benchmark pair. The wall-clock time (as opposed to CPU time) was measured for each run. The evaluation was performed on a computer with four octa-core Intel Xeon E5-4640 (2.4GHz) processors (in total 32 physical and 32 hyper-threading cores) with 512 GB of main memory. This is the highest level of parallelism used in the history of SAT competitions[8]. It should be noted, though, that the virtual cores provided by hyper-threading are not equivalent to physical cores, as they benefit only from wait states of a physical core to execute instructions of another thread. In fact, some solvers decided to use only 32 worker threads. We expect that for those solvers the authors experimentally determined that they do not profit from the additional virtual cores. Solvers using only 32 worker threads in the 2015 SAT Race were Plingeling and one version of Glucose (glucose-default). For the solvers pcasso-bb and pcasso it could not be determined how many worker threads they employed.

Three prizes were awarded in the Parallel Track based on the number of solved instances in the time limit of one hour per instance. The actual runtimes were not considered for the ranking.

6. Incremental Library Track

In the 2015 SAT Race we introduced a new track for solvers supporting interactive incremental SAT solving – the Incremental Library Track (ILT). The ILT differs significantly from all the other (traditional) tracks of the SAT competitions in the following aspects:

- In the traditional tracks we evaluate command line applications which take text files as input and produce text output. In the ILT we evaluate software libraries and communicate with them via function calls.
- The benchmarks for the traditional tracks are text files specifying a CNF formula. A benchmark for the ILT is an application (and inputs for it) that uses SAT solvers interactively.

In this section we first describe the motivation for the ILT and then give the detailed specification of the application program interface (API) that we designed and used to evaluate the incremental SAT solvers. The section is concluded by describing the benchmark programs used in the ILT (which also serve as demonstration of the API usage) and discussing the evaluation process of the ILT participants.

[8]The second place goes to the 2011 SAT Competition with 32 (physical) cores and the third to the 2009 SAT Competition with 16 cores.

6.1. Motivation

Many applications of SAT are based on solving a sequence of similar SAT formulas. The next formula in a sequence is usually generated by adding/removing a few clauses or variables to/from the previous formula. It is possible to solve such problems by solving the incrementally generated instances independently, however this might be very inefficient compared to an incremental SAT solver, which can reuse knowledge acquired while solving the previous instances (for example the learned clauses and variable activity statistics).

Most of the current state-of-the-art SAT solvers support incremental SAT solving, however each has its own interface, which differs from the others. That makes comparing them difficult. We believe it would be beneficial to have a single universal interface implemented by all the solvers supporting incremental SAT solving. Users who need to use incremental SAT solvers in their applications would strongly benefit from such a unified interface. Applications could be written without selecting a concrete incremental SAT solver in advance and migrating to a different solver would be just a matter of updating linking parameters.

6.2. The Proposed Interface - IPASIR

The name of the interface proposed for the 2015 SAT Race is IPASIR, which is the reversed acronym for "Re-entrant Incremental Satisfiability Application Program Interface[9]"

The interface was designed to have the following properties:

- Easy to implement by SAT solver developers.
- Easy to use, so that anyone can easily build SAT based applications (for industrial, scientific, educational or other purposes).
- Universal and powerful, i.e., usable in a broad range of applications.

The interface consists of nine C language functions which are inspired by the incremental interfaces of PicoSAT and Lingeling. The declarations of the functions along with short descriptions are displayed in Figure 1. The detailed specifications are given below.

The nine functions of the IPASIR interface can be split into two groups. The first group (service functions) contains four functions: `ipasir_signature`, `ipasir_init`, `ipasir_release` and `ipasir_set_terminate`.

ipasir_signature. The function `ipasir_signature` is used to identify the solver implementing the interface. The SAT solver is expected to return a C-style string containing its name and version.

[9] With an additional space and question mark it can also serve as an expression for offering a brewed beverage to a gentleman.

```
// Get solver name and version
const char* ipasir_signature();
// Initialize a solver instance and return a pointer to it
void* ipasir_init();
// Destroy the solver instance
void ipasir_release(void* solver);
// Set a callback function for aborting solving
void ipasir_set_terminate(void* solver, void* state,
                          int (*terminate)(void* state));
// Add a literal or finalize clause
void ipasir_add(void* solver, int lit_or_zero);
// Assume a literal for the next solve call
void ipasir_assume(void* solver, int lit);
// Solve the formula
int  ipasir_solve(void* solver);
// Retrieve a variables truth value (SAT case)
int  ipasir_val(void* solver, int lit);
// Check for a failed assumption (UNSAT case)
int  ipasir_failed(void* solver, int lit);
```

Figure 1: Declarations of the nine functions in the IPASIR interface.

ipasir_init. The purpose of the `ipasir_init` function is to create a new instance of the solver and return a pointer to it. This pointer is used as the first argument in all of the remaining seven functions.

ipasir_release. To destroy a solver, i.e., release all its resources and deallocate the memory it used we provide the `ipasir_release` function. The solver pointer cannot be used for any purposes after calling this function.

ipasir_set_terminate. The `ipasir_set_terminate` function can be used to specify a callback function which indicates whether the search should be aborted. The signature of the callback function is "`int terminate(void* state)`". It returns a non-zero value if the search is to be terminated. The solver is required to call this function periodically during the search process and abort it as soon as possible when a non-zero value is returned. The value of the parameter `state` in the calls of the callback function is identical to value received as the second argument of `ipasir_set_terminate`.

The second group of functions is used for SAT solving and contains the remaining five functions of the IPASIR interface. To better understand the semantics of these functions we define four states of the solver – UNKNOWN, SOLVING, SAT and UNSAT. After the `ipasir_init` call the solver is in the UNKNOWN state and is ready to receive input in the form of clauses and assumptions (see below). When the `ipasir_solve` function is called the state of the solver changes into SOLVING. When the search process is completed the

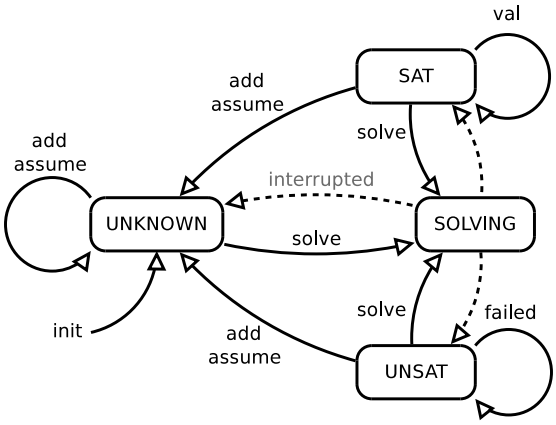

Figure 2: The four possible states of a SAT solver according to the IPASIR interface and the possible transitions between them. Solid edge transitions are explicitly associated to function calls while dashed transitions happen implicitly based on the result of the search process.

state changes to SAT or UNSAT depending on the result. The search might also get aborted (via returning a non-zero value in the callback function), then the state of the solver changes back to UNKNOWN. Figure 2 contains a diagram with the states and the allowed transitions between them. The label of each edge contains the names of the functions that can achieve the given transition.

ipasir_add. The formula to be solved is specified using the `ipasir_add` function. The clauses are added one literal at a time. Calling the `ipasir_add` function with a non-zero second argument adds a literal (specified by the argument) to the current clause. If the argument is zero, then the current clause is finalized and the next `ipasir_add` function call will initialize a new clause and add one literal to it. Literals are encoded as (non-zero) integers as in the DIMACS formats. They have to be smaller or equal to INT_MAX and strictly larger than INT_MIN (to avoid negation overflow). This applies to all the literal arguments in API functions. For example adding the clauses $(x_1 \vee \neg x_3)$ and (x_2) is achieved by the following sequence of calls: `ipasir_add(s, 1)`, `ipasir_add(s, -3)`, `ipasir_add(s, 0)`, `ipasir_add(s, 2)`, `ipasir_add(s, 0)`. Clauses added this way cannot be removed. The addition of removable clauses can be simulated using activation literals and assumptions[10]. The `ipasir_add` function may be called in any of the UNKNOWN, SAT and UNSAT states and it always results in the UNKNOWN state.

[10]Let $C_1, \ldots, C_m$ be clauses we wish to add and possibly remove later. Let $a_1, \ldots, a_m$ be a set of fresh (not used elsewhere) Boolean variables. We will add $(C_1 \vee a_1), \ldots, (C_m \vee a_m)$ and then run the solver with the assumptions $\neg a_1, \ldots, \neg a_m$. Removing any of the assumptions effectively removes the corresponding clause.

ipasir_assume. Assumptions in incremental SAT solving can be viewed as temporary unit clauses. Assumptions can be specified by calling the `ipasir_assume` function, which takes literals encoded as integers in the same way as the add function. All the added assumptions are valid only for the next `ipasir_solve` call. When the `ipasir_solve` call is finished all the assumptions are removed and a new set of assumptions can be specified. The removal of assumptions happens regardless of whether the search was aborted or completed. Similarly to `ipasir_add` the `ipasir_assume` function may be called in the UNKNOWN, SAT and UNSAT states and always results in the UNKNOWN state.

ipasir_solve. The most important function of the interface is `ipasir_solve`. It is used to solve the formula specified using the `ipasir_add` calls under the assumptions given by the `ipasir_assume` calls. When called, the solver changes to the SOLVING state until the formula is solved or the search is interrupted. If a satisfying assignment is found it returns the value 10 and the state of the solver is changed to SAT. In the case, that the problem is proven to be unsatisfiable, the function returns the value 20 and changes the state to UNSAT. If the search process was interrupted the return value is 0 and the solver returns to the UNKNOWN state. In each of the three cases the assumptions added before the `ipasir_solve` call are erased.

ipasir_val. In the case that `ipasir_solve` found a satisfying assignment and therefore the solver is in the SAT state we can call the `ipasir_val` function to retrieve the value of a variable (or literal). The return value of `ipasir_val(s,lit)` is $+lit$ if lit is true/satisfied under the solution and $-lit$ otherwise. The return value might be zero in the case that the truth value of the given variable (literal) is not assigned in the satisfying partial truth assignment. By calling `ipasir_val` for each variable we can retrieve the complete satisfying assignment. A call to `ipasir_add` or `ipasir_assume` invalidates the current solution and changes the state of the solver to UNKNOWN therefore calling `ipasir_val` is not allowed anymore.

ipasir_failed. If a formula is determined to be unsatisfiable under certain assumptions it is of interest to know which of the assumptions were actually used to prove the unsatisfiability. The conjunction of all used assumptions is already sufficient to prove unsatisfiability of the formula (see, e.g., [19]). This information can be retrieved by calling the `ipasir_failed` function using the assumption in question as the argument. The return value is 1 if the assumption was used and 0 otherwise. Analogously to `ipasir_val` the function `ipasir_failed` may only be called in the UNSAT state, i.e., between the call of `ipasir_solve` (which returned 20) and the first following call of `ipasir_add` or `ipasir_assume`.

The reader might wonder why is it allowed to call `ipasir_solve` from the SAT and UNSAT states. In the case of SAT it makes, indeed, no sense while the formula cannot change its satisfiability status without new input in the form of clauses or new assumptions from the user. However, in the case of UNSAT the unsatisfiability might be caused by the assumptions. Since after

each `ipasir_solve` call the assumptions are cleared the input has changed in a way that the formula might be satisfiable now.

We have described all the nine functions of the IPASIR interface. Examples of IPASIR usage can be found in Section 6.4 where we describe the benchmark applications used in the ILT.

6.3. Participation

For the participants of the ILT we prepared a package (ipasir.zip[11]) containing three SAT solvers (MiniSat, PicoSat and Sat4j) adapted to IPASIR and four simple IPASIR based applications for demonstration and benchmarking purposes. One of the adapted SAT solvers included in the package was the Java SAT solver Sat4j. We included Sat4j to demonstrate that although IPASIR is specified as a set of C functions it is possible to implement it in other languages than C/C++. The package also contains a simple Java binding and Java application to show how any IPASIR compatible SAT solver can be used in a Java application. The description of the example applications contained in the ipasir package, which were later also used as benchmark applications, can be found in the next subsection.

In order to participate in the ILT the participants were required to implement all nine functions (see Figure 1) of IPASIR following the specifications given in the previous section. The participants were asked to prepare a makefile that compiles their solver and generates a linux static library (.a file – object code archive file). Detailed instructions and examples on how to do this were included in the ipasir.zip package.

Eight solvers (solver versions) were submitted for the ILT, which we consider a very nice number considering that the ILT was organized for the first time. The list of participating solvers can be seen in Table 1.

6.4. Benchmarks

This section has two purposes. One is to describe the benchmark applications used in the ILT and the other is to demonstrate the usage of the IPASIR interface on simple examples.

In general, the benchmarks for the ILT consist of a command line application that solves some kind of a problem and a set of inputs for it. An example is a MaxSAT solver and a collection of MaxSAT instances.

We start with a very detailed description of the first ILT benchmark application – Essentials. The remaining three benchmarks are described only briefly. The source codes of all the applications can be found in the ipasir.zip package on the website of the 2015 SAT Race [20].

[11] `http://baldur.iti.kit.edu/sat-race-2015/downloads/ipasir.zip`

```
1  int pdr(int var) { return 2*var; }
2  int ndr(int var) { return 2*var - 1; }
3  int  dr(int lit) { return lit > 0 ? pdr(lit) : ndr(-lit); }
4
5  void Essentials(Formula f) {
6    void* s = ipasir_init();
7    for (int c = 0; c < f.clauses; c++) {
8      for (int k = 0; k < f.clause[c].size; k++) {
9        ipasir_add(s, dr(f.clause[c].lit[k]));
10     }
11     ipasir_add(s, 0);
12   }
13   for (int v = 1; v <= f.variables; v++) {
14     ipasir_add(s, -pdr(v));
15     ipasir_add(s, -ndr(v));
16     ipasir_add(s, 0);
17   }
18   for (int v = 1; v <= f.variables; v++) {
19     ipasir_assume(s, -pdr(v));
20     ipasir_assume(s, -ndr(v));
21     if (ipasir_solve(s) == 20) {
22       printf("%d is Essential\n", v);
23     } else {
24       printf("%d is not Essential\n", v);
25     }
26   }
27   ipasir_release(s);
28 }
```

Figure 3: Code of the *Essentials* benchmark application.

6.4.1. Essentials

Essentials is a detector of variables essential for satisfiability. For a satisfiable formula F and a variable x we say that x is essential for the satisfiability of F if x has to be assigned to True or False in each (partial) satisfying assignment of F.

Essentials is based on the so called dual-rail encoding of Boolean formulas [21] which works the following way. For each variable x occurring in a formula F we define two new variables x_T and x_F. These variables represent the fact that x is assigned to the value True (x_T) or False (x_F). If both x_T and x_F are False then x is unassigned, the case that they are both True is not allowed which can be expressed via binary clauses of the form $(\neg x_T \vee \neg x_F)$.

To construct the dual-rail encoding of a CNF formula F we replace each positive occurrence of x by x_T and each negative occurrence by x_F in all the

clauses of F for each variable x that occurs in F. Additionally we add a binary clause of the form $(\neg x_T \vee \neg x_F)$ for each variable (see previous paragraph). For example the dual rail encoding of the formula $(x \vee \neg y \vee z) \wedge (\neg x \vee \neg y) \wedge (\neg z)$ would be $(x_T \vee y_F \vee z_T) \wedge (x_F \vee y_F) \wedge (z_F) \wedge (\neg x_T \vee \neg x_F) \wedge (\neg y_T \vee \neg y_F) \wedge (\neg z_T \vee \neg z_F)$.

In Figure 3 we present the full code of the main function of the Essentials application. We assume that the input formula (which is satisfiable) is already parsed and we receive it as the parameter. The formula is represented as a struct containing the number of variables and clauses (as integers) and an array of clauses. Clauses are represented as structs containing the number of literals and an array of literals represented as integers[12].

In the first three lines we have helper functions computing the indices of the dual rail variables. The function `pdr` represents the translation of x to x_T and `ndr` the translation of x to x_F. The function `dr` translates literals by checking whether they are positive or negative and then calling `pdr` or `ndr` respectively.

The solver is initialized on line 6. On lines 7–12 we do the dual-rail encoding of the original clauses of the formula. On lines 13–17 we add the binary clauses ensuring that the variables cannot be both True and False at the same time.

The checking of the essentiality property happens on lines 18–26. For each of the variables we assume that they are not assigned. On line 19 we assume that the variable is not assigned to True and on 20 that not to False. Then we check the satisfiability of the formula under these assumptions. If the formula is unsatisfiable then the variable is essential for satisfiability. Note, that in the next iteration of the for loop the previous assumptions are removed.

The input of the Essentials application is CNF Boolean formulas – same as SAT solvers. In the 2015 SAT Race we selected the 50 easiest instances of the Main Track as benchmarks.

6.4.2. Partial MaxSAT Solver

The benchmark application PMaxSAT is an iterative strengthening based partial MaxSAT solver. A partial MaxSAT formula consists of two kinds of clauses – soft and hard. The goal of partial MaxSAT solving is to find a truth assignment that satisfies all the hard clauses and as many as possible soft clauses in a given formula.

The pseudo-code of PMaxSAT is displayed in Figure 4. The program returns the maximum number of soft clauses that can be satisfied while satisfying all the hard clauses. If the set of hard clauses is unsatisfiable it returns zero.

After the initialization of the solver we add the hard clauses using `ipasir_add` calls. The next step is to add the soft clauses. We add each soft clause with an extra variable called an activation variable. The activation variable is unique for each soft clause and does not appear anywhere in the original formula. Then we check the satisfiability status of the formula. At this point this is equivalent

[12] Positive literals are positive integers having the value of the variable, negative literals are represented as -value of the variable. For example x_3 is 3, $\neg x_7$ is -7, variables are numbered from 1.

```
1  int PMaxSAT(PMaxSatFormula f) {
2    void* s = ipasir_init();
3    add-hard-clauses(s, f);
4    add-soft-clauses-with-activation-literals(s, f);
5    int satSofts = 0;
6    while (ipasir_solve(s) == 10) {
7      satSofts = count-sat-soft-clauses(s, f);
8      add-cardinality-constraint(s, f, satSofts);
9    }
10   ipasir_release(s);
11   return satSofts;
12 }
```

Figure 4: Pseudo-code of the *PMaxSAT* benchmark application. The program determines how many soft clauses can be satisfied while satisfying all the hard clauses.

to checking that the hard clauses are satisfiable since all the soft clauses can be easily satisfied by setting all the activation literals to True. If the formula is satisfiable we count the number of soft clauses that are satisfied by at least one of their original literals, i.e., not by their activation variable. We save this value as the best solution so far and try to improve it. We do this by adding a cardinality constraint which says that the number of activation literals set to True must be strictly less than the best solution found so far. The cardinality constraint is translated into CNF and added using `ipasir_add` calls. For this translation we employ the PBLib Library [22]. After the cardinality constraint is added we solve the formula again. We repeat the cycle of solving and strengthening as long as the formula is satisfiable.

The PMaxSAT takes input in the form of WCNF files which is the standard input of MaxSAT solvers. For the ILT we used the 568 PMaxSat problems from the Industrial Track of the 2014 International MaxSat Competition.

6.4.3. Parallel Portfolio

The next benchmark application is a simplistic parallel portfolio SAT solver. The Portfolio application creates four instances of the incremental solver and adds all the input clauses to each of them. In order to achieve some degree of diversification the ordering of the clauses is randomly shuffled for each of the solver instances. A callback function for termination is added to each of the solver instances (by calling `ipasir_set_terminate`). The callback function is the same for each solver and all it does is return the value of a global variable which is initialized to zero. Any solver instance that solves the problem will set this variable to the value 1, which results in aborting the search by the still running solver instances. Finally four threads are created which call the `ipasir_solve` function of the corresponding solver instance.

This application is not really an incremental SAT application since it calls the solve function only once (in each thread). Nevertheless it is useful for testing

whether the solver is functioning properly and efficiently in a multithreaded environment.

The input of the Portfolio application is CNF Boolean formulas. As input we used the 100 instances from the Parallel Track of the SAT Race.

6.4.4. Incremental File Interpreter

The incremental file interpreter benchmark application is the only benchmark in the ILT that was not created by the organizers of the 2015 SAT Race. It was submitted by Florian Lonsing, Johannes Oetsch and Uwe Egly from the Vienna University of Technology.

The application is used to solve arbitrary sequences of formulas incrementally. For a given sequence of formulas, an input file describes how to update a formula to obtain its successor. The application then interprets these update instructions and translates them into IPASIR calls.

A detailed description of this approach, in particular the algorithm to generate update instructions based on a sequence of related formulas as well as the used file format, is presented in [23].

We used 50 input files which correspond to some benchmark problems used for the last Hardware Model Checking Competition (HWMCC 2014). Competition benchmarks are originally given as And-Inverter Graphs (AIGs). By means of the bounded model checking (BMC) based tool aigbmc, which is part of the AIGER package, CNFs are generated for checking AIGs incrementally. Each CNF corresponds to one unrolling step in BMC. An input file of the benchmarking application contains update instructions for the whole sequence of CNFs of one model checking problem. Therefore this benchmark application makes it possible to compare the incremental SAT solvers that are part of the IPASIR framework on the CNFs of the BMC workflow.

6.5. Evaluation

When announcing the ILT we stated that the winners will be determined based on the number of solved problems in a given time limit (like in the other tracks). However, after selecting the actual benchmarks this did not seem completely fair. For example the partial MaxSAT benchmark has much more input files than the other benchmarks and therefore it would influence the overall results more significantly. This is not fair, since different benchmark applications should contribute equally to the evaluation.

Our first attempt to solve this issue was to consider the relative number of solved inputs, i.e., the average percentage of solved inputs to the number of all inputs in each application. This approach is problematic if some of the applications contain inputs with high variance in the sense of solver performance. For example an application where the performance of the solvers is dissimilar has a much higher influence than an application where all the solvers solve almost the same number of problems.

Another way of evaluation, which resolves the issues mentioned above, is the ranking based comparison. We assign a rank to each solver for each application

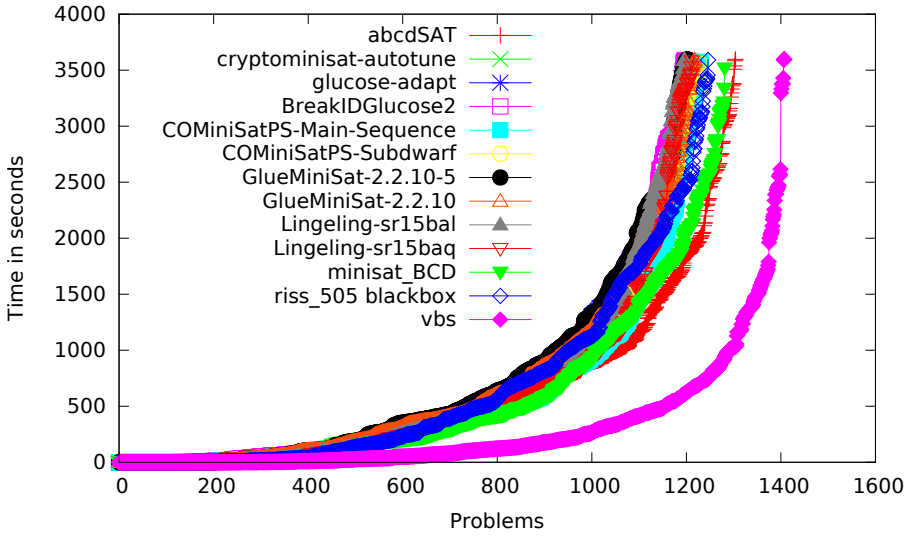

Figure 5: Runtimes of the top 12 solvers (and virtual best solver) in the Main Track for both SAT and UNSAT problems. The runtimes are aggregated from the five runs.

separately (based on the number of solved instances). The solvers are then compared based on their average rank. In this case each benchmark application has the same influence on the final ranking. Nevertheless, this ranking does not consider how much better or worse a given solver is compared to the others, which might be considered unfair.

When deciding how the first three prizes would be awarded in the ILT we considered several possible ranking methods and selected the solvers that ranked high in all of them. The final results are given in the next section. For future competitions the evaluation method should be fixed in the beginning.

The ILT evaluation was run on computers with two quad-core Intel Xeon X5355 (2.66 GHz, 8 cores in total) processors and 24 GB of main memory. The benchmark applications were executed in parallel, i.e., eight instances of a solver-application pair were running at the same time on eight different input files.[13] Two different solvers or two different applications were not allowed to run at the same time, therefore each solver-application pair could only interfere with itself (on different input files). The runtime limit for each application-solver-input triplet was 10 minutes.

[13]Due to cache and memory bandwidth limitations, the results will likely differ from a setting where each solver-application pair has access to the whole machine. Due to limited compute resources, we had to take this approach, though.

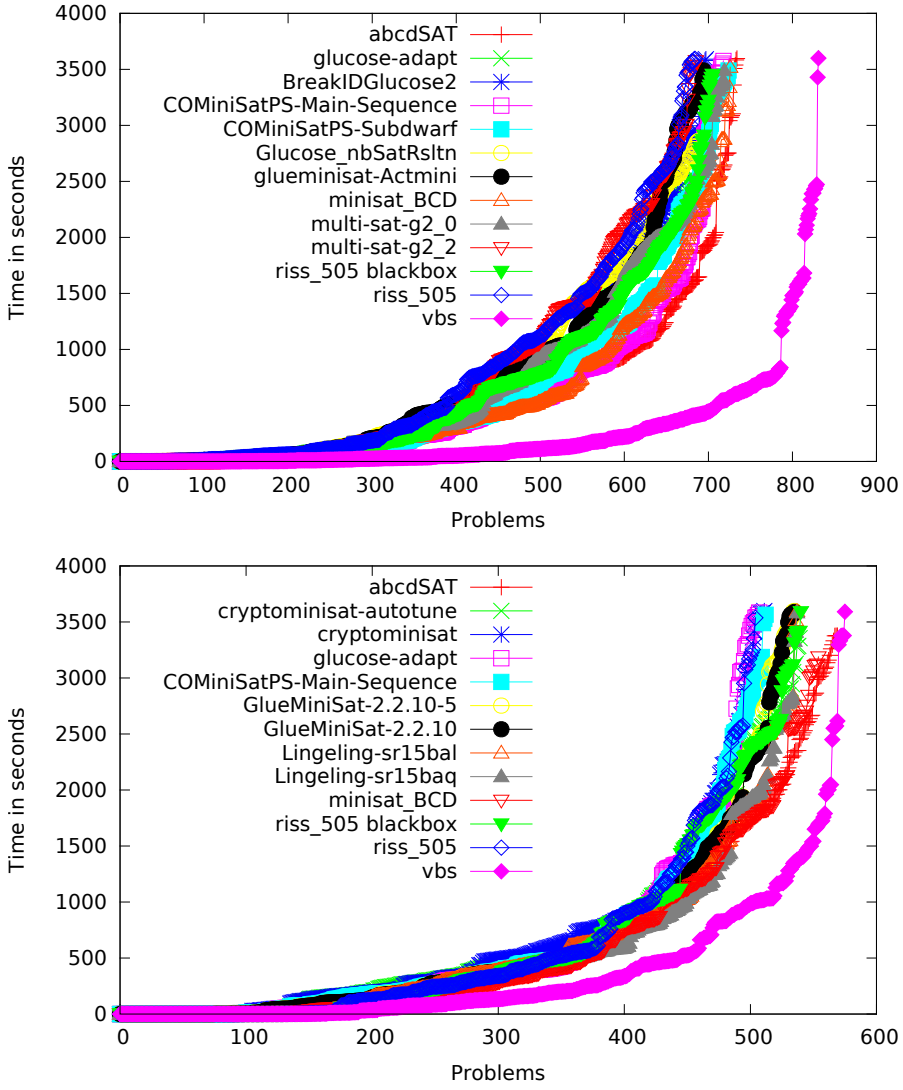

Figure 6: Runtimes of the top 12 solvers (and virtual best solver) in the Main Track separately for SAT (top) and UNSAT (bottom) problems. The runtimes are aggregated from the five runs.

All Instances			Satisfiable Instances			Unsatisfiable Instances		
solver name	**#s**	**avg.**	**solver name**	**#s**	**avg.**	**solver name**	**#s**	**avg.**
virtual best solver	284		virtual best solver	168		virtual best solver	116	
abcdSAT	261	261	abcdSAT	147	147	abcdSAT	114	114
minisat_BCD	257	256.4	minisat_BCD	146	145.4	minisat_BCD	111	111
riss-505-blackbox	256	249.4	COMiniSatPS-Subdwarf	145	145	riss-505-blackbox	109	108.2
COMiniSatPS-Main	247	246.4	multi-sat-g2-0	144	144	cryptominisat-autotune	108	108
COMiniSatPS-Subdwarf	246	245.8	COMiniSatPS-Main	144	143.8	Lingeling-sr15baq	108	107.6
GlueMiniSat-2.2.10	246	243.8	riss-505-blackbox	147	141.2	Lingeling-sr15bal	108	107.6
Lingeling-sr15baq	243	242.6	glueminisat-Actmini	139	139	GlueMiniSat-2.2.10	108	107.2
cryptominisat-autotune	242	241.8	BreakIDGlucose2	140	139.6	GlueMiniSat-2.2.10-5	108	107.2
GlueMiniSat-2.2.10-5	243	240.6	Glucose-nbSatRsltn	138	138	COMiniSatPS-Main	103	102.6
Lingeling-sr15bal	242	239.4	glucose-adapt	138	138	cryptominisat	103	102.4
BreakIDGlucose2	240	239.6	riss-505	138	137	riss-505	101	101
glucose-adapt	240	239.6	multi-sat-g2-2	138	137.2	glucose-adapt	102	101.6
riss-505	239	238	Glucose-nbSat	137	137	COMiniSatPS-Subdwarf	101	100.8
cryptominisat	238	237.4	glucose-community	138	137	CCAglucose2015	100	100
Glucose-nbSatRsltn	235	235	GlueMiniSat-2.2.10	138	136.6	BreakIDGlucose2	100	100
glucose-community	236	235	Lingeling-sr15baq	135	135	glucose-default	99	98.2
glueminisat-Actmini	236	234.8	cryptominisat	135	135	glucose-community	98	98
CCAglucose2015	235	232.6	Nigma-1.2.86	134	134	or-tools	97	97
multi-sat-g2-0	231	231	GlueMiniSat-2.2.10-5	135	133.4	Glucose-nbSatRsltn	97	97
Glucose-nbSat	232	231.8	glucose-default	134	133.4	Nigma-1.2.87	97	96.6
glucose-default	233	231.6	cryptominisat-autotune	134	133.8	glueminisat-Actmini	97	95.8
multi-sat-g2-2	231	229.8	CCAglucose2015	135	132.6	Nigma-1.2.86	95	94
Nigma-1.2.86	229	228	Lingeling-sr15bal	134	131.8	Glucose-nbSat	95	94.8
Nigma-1.2.87	227	226.6	Nigma-1.2.87	130	130	multi-sat-g2-2	93	92.6
dccaSatToRiss	215	214.4	dccaSatToRiss	125	124.4	dccaSatToRiss	90	90
or-tools	195	195	satUZK-seq	102	102	multi-sat-g2-0	87	87
satUZK-seq	165	164.8	or-tools	98	98	satUZK-seq	63	62.8
ratselfax-cnf-215	80	77.2	ratselfax-cnf-215	54	52	ratselfax-cnf-215	26	25.2

Table 2: The ranking of the solvers in the Main Track by the average number of solved instances (avg.) in the five runs. The results are given for all, satisfiable, and unsatisfiable instances. The column labeled #s contains the number of instances solved in at least one of the runs.

7. Competition Results

In this section we provide the results of the 2015 SAT Race. That includes the rankings and runtimes of the participating solvers in the different competition tracks and a summary of the technical novelties that have been implemented in the award winning solvers. We later analyze the results regarding trends. The detailed system descriptions provided by the authors of the participating solvers are available on the 2015 SAT Race website [20].

In the results for the Main and Parallel Tracks we included a special solver – the virtual best solver (VBS). The VBS (by definition) achieves the best result among the participating solvers for each benchmark. This hypothetical solver can be imagined as the perfect portfolio, which is able to select the best solver (among the participating solvers) for each benchmark in the competition in zero time.

7.1. Main Track Results

Table 2 shows the ranking of the solvers of the competition's Main Track. The ranking is based on the number of solved instances regardless of the satisfiability status. The first two prizes went to the solvers `abcdSAT` and `MiniSAT_BCD` written by Jingchao Chen from Donghua University in Shanghei (Sections 7.1.1 and 7.1.2) and the third price went to the solver `RISS` (Section 7.1.3) written by Norbert Manthey from the Technical University of Dresden.

The runtime distribution of the top 12 solvers is displayed in Figure 5 for all benchmarks and separately for SAT and UNSAT benchmarks in Figure 6. We can observe that the solvers are very close to each other. Only the winning solvers `abcdSAT` and `MiniSAT_BCD` stand out noticeably, especially on the UNSAT instances. The VBS outperforms all the solvers very significantly for the SAT problems, while for UNSAT problems it solves only a few extra instances (however with better runtimes).

7.1.1. abcdSAT

The solver `abcdSAT` is built upon `Glucose 2.3` and uses `Lingeling 587f` as pre-processor. Major improvements are an extended Fourier-Motzkin based variable elimination procedure that is based on cardinality constraint detection of up to $^{\leq}4$-constraints. Previous approaches applied this kind of formula simplification in conjunction with detection of up to $^{\leq}2$-constraints [17]. Furthermore they introduced a fast and bounded clause-based implementation of Hyper Binary Resolution [24].

`abcdSAT` introduces an original approach of using the results of a preceding phase of Blocked Clause Decomposition (BCD) [25] in CDCL search. For decision levels 1 to 3 the original `pickBranchLit` method is overridden and a BCD-based decision policy is used instead. Using the large blocked set and the clause order that is imposed by their possible elimination [26] they use the first decision literal (at level 0) as root and try to satisfy the adjacent blocked clauses in their reverse elimination order. This localizes search in some sense regarding possibly functional variable relations [27].

7.1.2. Minisat_BCD

`Minisat_BCD` incorporates all the advances of `abcdSAT` (Section 7.1.1). The difference lies in the applied learned clause database management policy. While `abcdSAT` uses the clause management of `Glucose 2.3`, `Minisat_BCD` applies the hybrid policy of `MiniSat_HACK_999ED` which has been very successful in the past and is now introduced in several shapes in other winning solvers of the 2015 competition. As a clause's LBD (Glue) value has been shown to be a good quality measure for learned clauses [28] only the clauses with an LBD value up to the threshold of 5 are kept permanently in the database. To establish a kind of short-term memory and to break ties another pool of clauses is managed by the original Minisat activity-based heuristic [29].

7.1.3. RISS

`RISS` is built upon `MiniSAT` and also incorporates the improvements that have been introduced with `Glucose 2.2`. An important part of `RISS` is its pre-processor `Coprocessor` that can perform recent formula simplification techniques. `Coprocessor` is used for preprocessing as well as inprocessing [6].

`RISS` now includes a form of *restricted extended resolution* with gate-recognition based clause minimization techniques [30, 31]. Its learned clause removal strategy is an individual combination of the `Glucose 2.2` LBD-based and the `Minisat 2.2` activity-based strategy.

`Coprocessor` is now capable of adding *resolution asymmetric tautologies (RAT)* [6] that subsume other clauses. Such subsuming clauses can also be *blocked clauses* and *covered clauses*.

`RISS` uses an *automatic configuration* routine that projects 382 formula features to 40 eigenvectors by application of Principal Component Analysis (PCA). Based on the features of the formula one of the 40 best configurations is selected using a k-nearest neighbor algorithm.

7.2. Parallel Track

The overview of the Parallel Track results is provided in Table 3. The first prize went to the solver `Glucose 4` (Section 7.2.1) written by Gilles Audemard from the University of Lille-Nord and Laurent Simon from the University of Bordeaux in France. The second and third prizes went to `plingeling` and `treengeling` (Sections 7.2.2 and 7.2.3) written by Armin Biere from the Johannes Kepler University in Linz, Austria.

The distribution of runtimes (wall-clock time) is plotted in Figure 7. As in the Main Track the VBS is a clear winner. By looking at the number of solved instances in Table 3 we can see that this is again mostly due to the satisfiable instances. As for the other solvers we can notice that they are split into two clusters – five solvers solving over 62 instances and six solvers solving less than 38 instances. Five solvers in low-performance cluster perform particularly bad on unsatisfiable instances. This could be due to having set the restrictions on clause-sharing too strong or due to performing too little restarts [32]. In contrast to the other five solvers in the low-performance cluster the parallel

	All		Satisfiable		Unsatisfiable	
	solver name	**#s**	**solver name**	**#s**	**solver name**	**#s**
0	virt-best-solver	86	virt-best-solver	53	virt-best-solver	33
1	glucose-adapt	78	treengeling	47	glucose-adapt	32
2	plingeling	73	glucose-adapt	46	plingeling	28
3	treengeling	73	plingeling	45	glucose-default	28
4	cbpenelope	66	cbpenelope	43	treengeling	26
5	glucose-default	62	satuzk	34	crypto–auto	26
6	satuzk	38	glucose-default	34	crypto	25
7	pcasso-bb	37	pcasso-bb	29	cbpenelope	23
8	pcasso	37	pcasso	29	pcasso-bb	8
9	crypto–auto	32	paraglueminisat	25	pcasso	8
10	paraglueminisat	29	crypto–auto	6	satuzk	4
11	crypto	29	crypto	4	paraglueminisat	4

Table 3: The ranking of the solvers in the Parallel Track based on the number of solved instances (#s). The results are given for all, satisfiable, and unsatisfiable instances.

versions of `CryptoMinisat` are highly competitive on unsatisfiable problems (even among the top three solvers) but perform really bad on the given satisfiable instances. By checking the results in the Main Track (table 2) we can see that already the sequential version of the solver shows a similar performance gap. So `CryptoMinisat` seems to be specifically tuned to find proofs in contrast to finding counter-examples.

7.2.1. Glucose 4

The winner of the Parallel Track is the *adaptive* version of `Glucose 4`. Most winning solvers have adopted `Glucose`'s LBD-based [28] ultra-rapid restarts and a lean clause database management. While other authors use hybrid strategies to balance the performance of their solver on unsatisfiable and satisfiable instances (see [32] for details), the authors of `Glucose` decided to introduce a dynamic adaptive strategy that is capable of switching heuristics online.

In contrast to other *adaptive* approaches that focus on problem feature extraction `Glucose 4` records its runtime parameters (while running in default configuration for a constant amount of conflicts) which then are used as indicators for a strategy switch.

The parallel version of `Glucose` a.k.a. `Glucose Syrup` uses a lazy clause sharing approach [33] that shares only clauses that have been seen at least twice in recent conflicts and imposes common restrictions on their size and LBD-value. `Glucose` limits clause sharing on 32 cores even more in order to satisfy memory constraints. Only half of the cores use the adaptive strategy while the others stick to the default configuration.

7.2.2. Plingeling

`Plingeling` [34] is based on the award winning solver `Lingeling` [34] that now knows 13 different optimized parameter settings for different buckets or

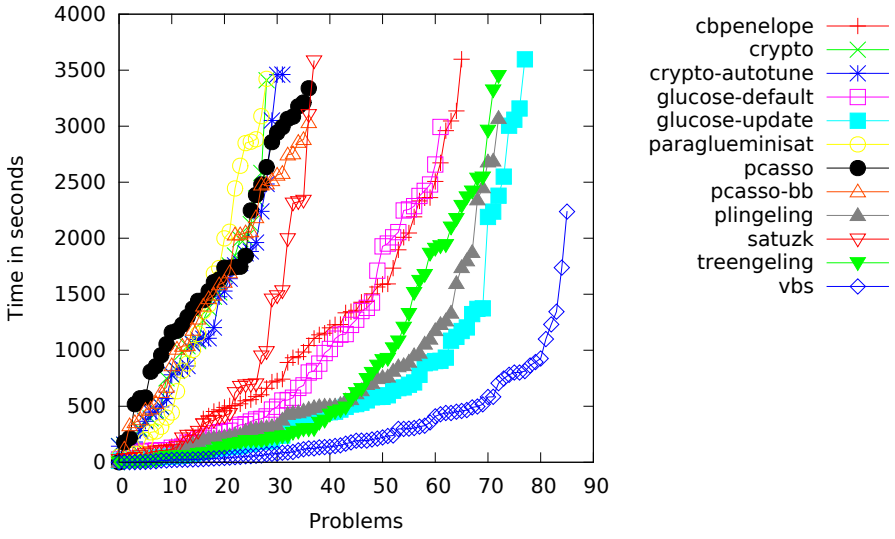

Figure 7: Runtimes (wall-clock time) of the solvers participating in the Parallel Track.

families of problems. While in the sequential version the problem at hand is classified by a k-nearest neighbor algorithm the parallel version runs all of these experimentally determined "best" configurations.

`Plingeling` traditionally shares unit clauses and equivalences. Since 2013 also clauses with low LBD values are shared. Clauses containing eliminated variables are rejected by the instances.

7.2.3. Treengeling

`Treengeling` [34] is based on the "Cube & Conquer" idea [35]. It uses look-ahead for splitting the CNF instance into multiple independent sub-problems. Those are simplified independently and split again. As soon as a sub-problem becomes simple enough to be solved more efficiently by CDCL solving, the original idea was to switch to CDCL solving. Treengeling improves on that by using "Concurrent Cube & Conquer" [36], which interleaves CDCL search with look-ahead in parallel. Actually, the simplest unsolved sub-problems are all split (using look-ahead and cloning), simplified (through preprocessing) and searched (CDCL) in parallel, until all sub-problems are proved to be unsatisfiable or one satisfiable sub-problem is found. In addition few parallel instances of Lingeling are started using the call-back infrastructure existing in Lingeling to implement Plingeling. These parallel solvers interact with the "Cube & Conquer" part in a Portfolio style manner, exporting unit clauses and importing the negation of unsatisfiable cubes. Otherwise there is no sharing of learned clauses among "Cube & Conquer" solver instances. For the parallel portfolio solvers the integration of the local search solver YalSAT [34] is enabled, which is run in intervals. Treengeling is expected to work very well for hard combinatorial

	Essential		HWMC		Portfolio		P-MaxSat	
	solver	**#s**	**solver**	**#s**	**solver**	**#s**	**solver**	**#s**
1	glucose	48	crypto	1454	comin-earth	12	crypto-auto	271
2	crypto	48	crypto-auto	1452	satuzk	5	crypto	266
3	crypto-auto	47	comin-sun	1434	picosat961	5	glucose	259
4	comin-sun	45	glucose	1407	comin-sun	5	comin-sun	250
5	comin-earth	45	comin-earth	1406	riss 505	4	riss 504	244
6	riss 505	44	riss 505	1372	riss 504	2	comin-earth	244
7	riss 504	44	riss 504	1370	glucose	1	riss 505	234
8	picosat961	44	picosat961	1285	crypto-auto	0	satuzk	204
9	satuzk	43	satuzk	842	crypto	0	picosat961	165

Table 4: The ranking of the Incremental Library Track solvers based on the number of solved instances (#s) for each benchmark application separately.

problems, where sharing of learned clauses is less useful.

7.3. Incremental Library Track

As discussed in Subsection 6.5 the evaluation criteria for the ILT are not as simple as for the other two tracks and there are several alternative ways to define them. Since the exact evaluation criteria were not announced ahead the actual selection of the winning solvers was done "manually" by the organizers with the goal to be as fair as possible.

Firstly, we decided that different versions of a solver will be considered as one solver. Then, by looking at the the rankings of the solvers for each benchmark separately (displayed in Table 4), we decided that the prizes will be distributed between the solvers `CoMinisatPS`, `CryptoMiniSat` and `Glucose`. The reason is that the remaining solvers never appear between the top three solvers for any of the benchmarks (the solver `picosat` was entering hors concours).

Based on the total number of solved problems, percentage of problems solved and relative ranking in each benchmark category we decided to award two first prizes to the solvers `CoMinisatPS` and `CryptoMiniSat` and the third prize to the solver `Glucose`.

The solver `CoMinisatPS` was submitted by Chanseok Oh from New York University, USA. The authors of `CryptoMiniSat 4.4` are Mate Soos and Marius Lindauer. `Glucose` is developed by Gilles Audemard from the University of Lille-Nord and Laurent Simon from the University of Bordeaux in France. Brief descriptions of the winning solvers follow.

7.3.1. CoMinisatPS

`CoMiniSatPS` comes with a *three-tiered learned clause database management* that was first described by its author in [32]. The strategy keeps a core of learned clauses with extremely low LBD value (1-3) and manages clauses with slightly higher but still low LBD (4-6) in a so called mid-tier database that keeps only recently used clauses. A constant amount of clauses is managed by the classical `MiniSat` [29] clause activity heuristic.

`CoMiniSatPS` applies a *hybrid restart strategy* with alternating phases of Luby-style restarts [37, 29] and `Glucose`-style ultra-rapid restarts [28]. In the ultra-rapid phase a lower *variable decay factor* is used than in the Luby-phase, thus respecting the finding that ultra-rapid restarts and lower variable decay factors both correlate positively with high performance on unsatisfiable problems [32] and vice versa.

Specific to the incremental version of `CoMiniSatPS` is its incremental variable elimination procedure that has already been described in [38]. At the end of each incremental run a phase of extreme clause learning is started and a majority of the learned clauses that accumulated in the database during the solver run is cleared.

7.3.2. CryptoMiniSat 4.4

`CryptoMiniSat` now comes with hybrid clause database management like it was introduced by the solver `SWDiA5BY` which is the predecessor of `CoMinisatPS` (Section 7.3.1), i.e. it keeps a core of clauses with low LBD value and manages a constant amount of recently learned clauses by MiniSat's activity score. Also like in `SWDiA5BY` a `Glucose`-style restart strategy is used (with LBD-based ultra-rapid restarts in conjunction with restart blocking phases).

`CryptoMiniSat` is an inprocessing SAT solver [6] and the new version comes with optimized data-structures for that, specifically the watcher data structure is used to store occurrence lists and data related to recognized gates.

In order to make `CryptoMiniSat` adaptive, the top 5 parameter configurations were experimentally determined and then a classifier has been trained with benchmark problems from the previous SAT Competitions to select the best configuration for the problem at hand. The resulting decision tree is compiled into the `CryptoMiniSat` source code.

7.3.3. Glucose 4

For a description of the base version see Section 7.2.1. The incremental version introduced special handling of so called selector literals (due to assumption-based clause removal) [19]. As selector literals could not be known in advance in the competition scenario, these optimizations specific to the incremental version of `Glucose` were ineffective in the competition.

8. Analysis of Results

8.1. Essential Improvements in Most Winning Solvers

One major result of the competition is that hybrid heuristics have established themselves widely among the best performing solvers, thus providing a trade-off between heuristics that perform well on unsatisfiable problems and those that perform well on satisfiable problems [32].

One of these hybrid approaches is lean clause database management that is based on the learned clauses LBD-values, which is combined with another clause-database of constant size that is managed by the classic activity-based cleanup

procedure. Another widely adopted hybrid approach is the rapid Luby [37] style restarts in combination with the ultra-rapid restarts based on LBD-values of learned clauses.

Most winning solvers perform a kind of automatic adaption of heuristics based on automatic feature extraction of the problem at hand or using runtime parameters.

One novelty in this competition was introduced by the winner of the Main Track, `abcdSAT` (Section 7.1.1), that makes creative use of blocked clause decomposition to restrict variable selection for a constant amount of decision levels to a set of variables that satisfies clauses that are adjacent in the order imposed by blocked clause elimination. The success of this method might be due to the strong connection between blocked clauses and functional dependencies among variables [27] and it shows that structure based solving can elevate performance of SAT solvers considerably.

8.2. Similarity of Solvers

We evaluated the similarity of the solvers using the notion of Spearman's rank correlation coefficient [39]. The basic idea of this comparison is that two solvers are similar if the same instances are easy (resp. hard) for both. More precisely, if we sort the list of benchmarks for solver a based on the runtimes required to solve them and do the same for solver b then the more similar these lists are the more similar are the solvers a and b.

Spearman's rank correlation coefficient is defined as:

$$\rho = 1 - \frac{6 \sum (r_a - r_b)^2}{n(n^2 - 1)} \tag{1}$$

where n is the size of the sample (in our case the number of benchmarks) and r_a,r_b are pairs of ranks for a given benchmark and two solvers a and b. The sum is over the set of benchmarks.

Let us consider the following simple example. We compare three solvers a, b and c on five benchmarks $b_1, \ldots, b_5$. The measured runtimes of each solver for each benchmark are displayed in Table 5 (columns t_a, t_b and t_c). Based on these runtimes we assign the ranks of the benchmarks for each solver (columns r_a, r_b and r_c). Next, using Equation 1, we compute for each pair of solvers the sum of the squares of rank differences for each benchmark. These values are multiplied by 6, divided by 120 ($= 5(5^2 - 1)$) and subtracted from 1 to obtain the resulting correlation coefficient ρ. In our example solvers b and c are the most similar to each other while c and a are the most different.

We calculated Spearman's rank correlation coefficient (ρ) for all the solver pairs in the Main and Parallel Tracks. These results are displayed in Figure 8 and Figure 9 in the form of heat-maps. The figures also contain a dendrogram to illustrate the arrangement of solver clusters produced by hierarchical clustering based on ρ.

By looking at the Main Track solvers (Figure 8) we can observe high similarity mostly between the different versions of the same solver (`lingeling`,

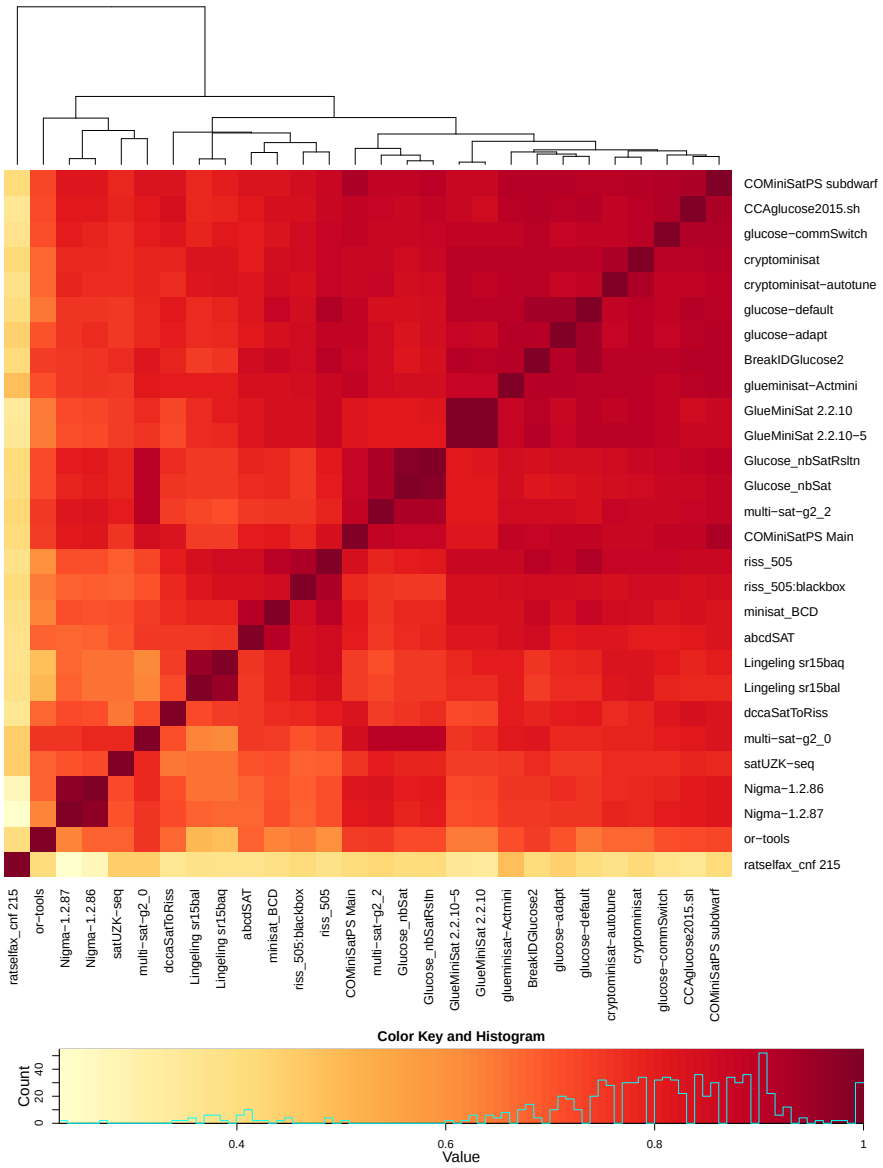

Figure 8: Heat-map and dendrogram based on the similarity of the solvers participating in the Main Track. Similarity is defined as Spearman's rank correlation coefficient. Darker regions mean that the solvers are more similar.

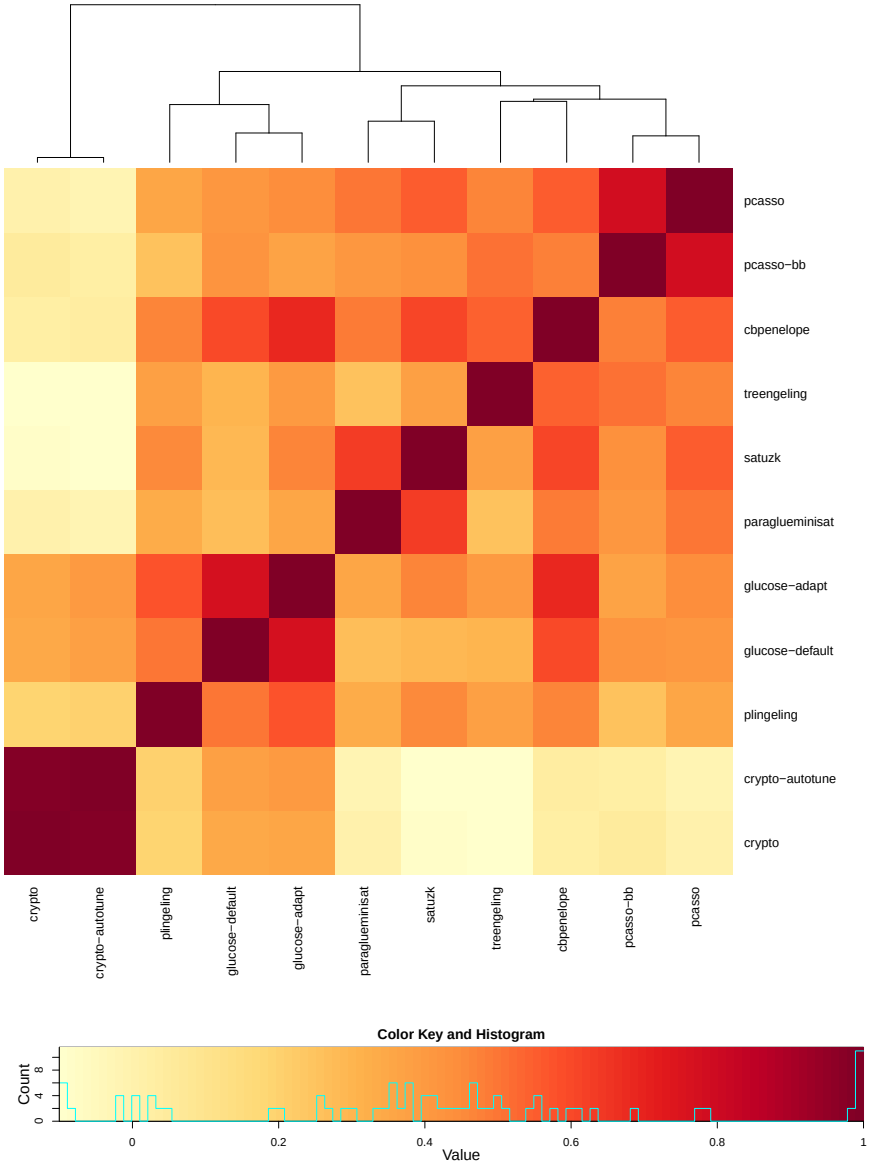

Figure 9: Heat-map and dendrogram based on the similarity of the solvers participating in the Parallel Track. Similarity is defined as Spearman's rank correlation coefficient. Darker regions mean that the solvers are more similar.

	solv. a		solv. b		solv. c		a vs. b	b vs. c	c vs. a
	t_a	r_a	t_b	r_b	t_c	r_c	$(r_a - r_b)^2$	$(r_b - r_c)^2$	$(r_c - r_a)^2$
b_1	0.1	1	0.4	3	0.7	4	4	1	9
b_2	0.3	2	0.2	2	0.3	2	0	0	0
b_3	0.5	3	0.1	1	0.1	1	4	0	4
b_4	0.7	4	0.9	5	0.8	5	1	0	1
b_5	0.9	5	0.8	4	0.5	3	1	1	4
						$\rho =$	0.50	0.90	0.10

Table 5: Spearman's rank correlation coefficient computation example.

`GlueMiniSat`, `Nigma`) or solvers from the same author (the winning solvers `MiniSatBCD` and `abcdSAT` by Jingchao Chen). Actually the two versions of `GlueMiniSat` have $\rho = 1$, which means they are perfectly monotonically related. The solver most different from all the others is `Ratselfax` followed by `or-tools`. Both these solvers are more general combinatorial optimization tools with SAT solving being just one of the problems they can be used for. Overall, the solvers participating in the Main Track are very similar to each other with an average ρ of 0.786 (average over all pairs).

One possible explanation for this high similarity of SAT solvers is that it is a negative side-effect of SAT competitions, in particular their benchmark selection method. The benchmarks of a competition are selected to be similar to previous competitions and therefore the solvers are trying to be similar to previous winning solvers. Indeed, most of the solvers are built on top of these successful solvers such as `Glucose`, which itself is based on `MiniSat`. The benchmark selection methods should be revised to address this issue.

On the other hand, the solvers participating in the Parallel Track (Figure 9) are much less similar with an average ρ of 0.411. High similarity is again observed between different versions of the same solvers (`CryptoMiniSat`, `glucose`, `pcasso`). Surprisingly the outliers of this track are the two versions of `CryptoMiniSat`. The correlation coefficient of both `CryptoMiniSat` versions is actually negative with `SatUZK` and `Treengeling`.

The decreased similarity of the solvers in the Parallel Track might be the result of introducing a new execution environment (in the sense of relatively high parallelism, i.e., 64 cores compared to at most 16 cores used in previous competitions).

8.3. Stability of Performance

Thanks to the fact, that we performed five runs of each solver on each benchmark in the Main Track we can now analyze the difference in the performance of the solvers across these five runs.

For each benchmark and solver we took the five runtime values and computed their standard deviation (*sd*) and the more robust median absolute deviation

Solver Name	All		Sat		Unsat	
	sd	mad	sd	mad	sd	mad
abcdSAT	11.8	**7.1**	**9.1**	**5.6**	15.3	9.0
cryptominisat	17.8	9.9	17.0	9.1	18.8	11.0
cryptominisat-autotune	17.7	9.7	16.0	9.1	19.8	10.6
dccaSatToRiss	21.4	13.5	22.2	15.8	20.1	10.3
glucose-adapt	17.4	11.2	17.8	11.6	16.9	10.6
glucose-community-switching	15.7	8.7	17.1	9.4	13.6	7.9
BreakIDGlucose2	17.7	10.4	18.2	11.5	17.1	8.8
CCAglucose2015	13.9	8.6	15.6	9.6	**11.6**	7.3
COMiniSatPS-Main	15.9	9.3	15.9	9.1	16.0	9.7
COMiniSatPS-Subdwarf	14.7	9.2	15.1	9.6	14.2	8.7
glucose	16.0	9.9	18.2	11.6	13.1	7.8
Glucose_nbSat	19.4	12.3	18.3	12.5	21.1	12.1
Glucose_nbSatRsltn	19.5	12.7	19.9	13.6	18.9	11.4
GlueMiniSat2.2.10	18.2	10.8	17.9	11.1	18.6	10.5
GlueMiniSat2.2.10-5	16.4	10.2	17.2	11.2	15.5	8.9
glueminisat-Actmini	22.7	12.3	28.0	13.5	15.0	10.6
Lingeling-sr15bal	14.8	8.9	16.6	9.9	12.5	7.6
Lingeling-sr15baq	13.9	8.6	15.6	10.3	11.8	**6.6**
MiniSatBCD	**10.9**	7.2	10.0	6.5	12.1	8.1
multi-sat-g2_0	18.4	11.3	17.1	10.5	20.6	12.6
multi-sat-g2_2	20.1	13.6	18.6	12.2	22.4	**15.6**
Nigma-1.2.86	21.3	12.6	20.3	11.8	22.6	13.9
Nigma-1.2.87	22.2	12.1	20.3	10.2	24.7	14.6
or-tools	21.9	12.8	19.1	11.7	**25.8**	14.3
Ratselfax 215	25.5	**16.1**	30.3	**18.8**	14.5	10.1
riss_505-blackbox	**94.2**	12.0	**152.7**	13.8	15.4	9.5
riss_505	18.7	11.8	20.3	12.7	16.5	10.5
satUZK-seq	13.6	8.6	11.1	7.5	17.8	10.4

Table 6: Statistical dispersion (or variation) of the runtimes for the five runs of each solver from the Main Track. The table shows the average standard deviation (sd) and average median absolute deviation (mad) for all the benchmark solved in at least one of the runs. The minimum and maximum values of each column are highlighted.

(*mad*), which are defined as:

$$sd = \sqrt{\sum_{i=1}^{n} \frac{(\mu - t_i)^2}{n}} \qquad mad = \text{median}_{i=1\ldots n}(|M - t_i|) \tag{2}$$

where n is the number of values (in our case five) while μ is the average and M the median of the runtime values.

Table 6 contains the average *sd* and *mad* over all the benchmarks. We can observe, that the two winning solvers (`abcdSAT` and `MinisatBCD`) have very stable performance throughout the five runs while the performance of `Riss - blackbox` (which finished as third) is highly variable. Actually, for the satisfiable instances `abcdSAT` is the most stable solver while `Riss - blackbox` is the least. Based on the results of `abcdSAT` and `MinisatBCD` we can draw a positive conclusion that it is possible to have SAT solvers that consistently deliver high performance.

9. Conclusion

Competitions of Boolean satisfiability (SAT) solvers have been organized regularly since 2002 and they serve as the main tool for evaluating and comparing state-of-the-art SAT solvers. We believe they also serve as one of the main motivators for researchers and developers to continually invent new algorithms and implementation techniques for SAT solving.

The aim of the 2015 SAT Race was to continue the tradition of SAT solving competitions and also to introduce new challenges. The most significant novelty of the 2015 SAT Race is the Incremental Library Track (ILT). For the ILT we have designed an API (called IPASIR) that makes it possible to develop SAT solving based applications without committing to any specific SAT solver. We believe that this will prove to be very useful for SAT solver users and increase the already high number of systems which use SAT solvers as their reasoning engines. We hope that the ILT Track will be continued in the following competitions and the IPASIR interface (or its extension) will get widely accepted and adopted by the SAT community.

In this paper we provided an overview of the 2015 SAT Race. We gave a detailed description of the new Incremental Track and the IPASIR interface. We presented and analyzed the results of the competition and provided an overview of the winning solvers and the new techniques they use.

9.1. Lessons Learned

The solver similarity results indicated, that the solvers are very similar to each other. This could be a negative side effect of SAT competitions and their benchmark selection methodology. Future benchmark selection methods should be designed while keeping this in mind.

The ILT had only one benchmark submission. Since there are many more systems based on incremental SAT solving we assume that our call for benchmarks did not reach the developers. Until the ILT becomes more traditional these developers should be contacted directly.

The evaluation criteria for the ILT were not clearly defined before the competition. For future competitions we must design a fair and well defined evaluation mechanism and publish it in the call for participation.

Acknowledgments

This research was partially supported by DFG project SA 933/11-1 and by FWF, NFN Grant S11408-N23 (RiSE).

References

[1] H. A. Kautz, B. Selman, et al., Planning as satisfiability., in: European Conference on Artificial Intelligence (ECAI), Vol. 92, 1992, pp. 359–363.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, M. K. Ganai, Robust boolean reasoning for equivalence checking and functional property verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21 (12) (2002) 1337–1394.

[3] C. Flanagan, R. Joshi, X. Ou, J. B. Saxe, Theorem proving using lazy proof explication, in: Computer Aided Verification (CAV), Springer, 2003, pp. 355–367.

[4] S. A. Cook, The complexity of theorem-proving procedures, in: ACM Symposium on Theory of Computing, ACM, New York, NY, USA, 1971, pp. 151–158.

[5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: Proceedings of the 38th annual Design Automation Conference, ACM, 2001, pp. 530–535.

[6] M. Järvisalo, M. J. H. Heule, A. Biere, Inprocessing rules, in: International Joint Conference on Automated Reasoning (IJCAR), Vol. 7364 of LNCS, Springer, 2012, pp. 355–370.

[7] M. Buro, H. K. Büning, Report on a SAT competition, Fachbereich Math.-Informatik, Univ. Gesamthochschule, 1992.

[8] D. S. Johnson, M. A. Trick, Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993, Vol. 26, American Mathematical Soc., 1996.

[9] M. Järvisalo, D. Le Berre, O. Roussel, L. Simon, The international SAT solver competitions, AI Magazine 33 (1) (2012) 89–92.

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

[11] A. Biere, M. Heule, H. van Maaren, T. Walsh, Conflict-driven clause learning SAT solvers, Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications (2009) 131–153.

[12] A. Stump, G. Sutcliffe, C. Tinelli, Starexec: A cross-community infrastructure for logic solving, in: S. Demri, D. Kapur, C. Weidenbach (Eds.), Automated Reasoning, Vol. 8562 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 367–373.

[13] G. Katsirelos, A. Sabharwal, H. Samulowitz, L. Simon, Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers, in: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, AAAI'13, AAAI Press, 2013, pp. 481–488.

[14] S. Hölldobler, N. Manthey, A. Saptawijaya, Improving resource-unaware SAT solvers, in: Logic for Programming, Artificial Intelligence, and Reasoning, Springer, 2010, pp. 519–534.

[15] R. Greenlaw, H. J. Hoover, W. L. Ruzzo, Limits to parallel computation: P-completeness theory, Vol. 200, Oxford university press Oxford, 1995.

[16] Y. Hamadi, S. Jabbour, L. Sais, Manysat: a parallel SAT solver, in: Satisfiability, Boolean Modeling and Computation, Vol. 6, 2008, pp. 245–262.

[17] A. Biere, Lingeling, plingeling and treengeling entering the SAT competition 2013., in: Proceedings of SAT Competition 2013, University of Helsinki, 2013, pp. 51–52.

[18] M. Järvisalo, D. Le Berre, O. Roussel, The SAT 2011 Competition – Results of Phase 1 - slides, `http://www.cril.univ-artois.fr/SAT11/phase1.pdf`, accessed: 2015-12-18 (2011).

[19] G. Audemard, J.-M. Lagniez, L. Simon, Improving glucose for incremental SAT solving with assumptions: Application to mus extraction, in: Theory and Applications of Satisfiability Testing (SAT), Berlin, Heidelberg, 2013, pp. 309–317.

[20] T. Balyo, M. Iser, C. Sinz, SAT Race 2015 Website, `http://baldur.iti.kit.edu/sat-race-2015`, accessed: 2015-11-11 (2015).

[21] R. E. Bryant, Boolean analysis of mos circuits, in: IEEE Transactions on Computer Aided Design, Vol. 6, 1987, pp. 634–649.

[22] T. Philipp, P. Steinke, PBLib – a library for encoding pseudo-boolean constraints into CNF, in: Theory and Applications of Satisfiability Testing (SAT), Vol. 9340 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 9–16.

[23] U. Egly, F. Lonsing, J. Oetsch, Automated benchmarking of incremental sat and qbf solvers, in: Logic for Programming, Artificial Intelligence, and Reasoning, Springer, 2015, pp. 178–186.

[24] J. Chen, Minisat bcd and abcdsat: Solvers based on blocked clause decomposition, `http://baldur.iti.kit.edu/sat-race-2015/descriptions/abcdSAT_Minisat_BCD.pdf`, accessed: 2015-12-12 (2015).

[25] M. J. H. Heule, A. Biere, Blocked clause decomposition, in: Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), 2013, pp. 423–438.

[26] M. Järvisalo, A. Biere, M. J. H. Heule, Blocked clause elimination, in: J. Esparza, R. Majumdar (Eds.), International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Vol. 6015 of LNCS, Springer, 2010, pp. 129–144.

[27] M. Iser, N. Manthey, C. Sinz, Recognition of nested gates in CNF formulas, in: Theory and Applications of Satisfiability Testing (SAT), 2015, pp. 255–271.

[28] G. Audemard, L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: International Joint Conference on Artifical Intelligence (IJCAI), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009, pp. 399–404.

[29] N. Eén, N. Sörensson, An extensible sat-solver, in: Theory and Applications of Satisfiability Testing (SAT), 2003, pp. 502–518.

[30] G. Audemard, G. Katsirelos, L. Simon, A restriction of extended resolution for clause learning SAT solvers, in: Conference on Artificial Intelligence (AAAI), 2010, pp. 15–20.

[31] N. Manthey, Extended resolution in modern SAT solving, in: Joint Automated Reasoning Workshop and Deduktionstreffen, 2014, pp. 26–27.

[32] C. Oh, Between SAT and UNSAT: the fundamental difference in CDCL SAT, in: Theory and Applications of Satisfiability Testing (SAT), 2015, pp. 307–323.

[33] G. Audemard, L. Simon, Lazy clause exchange policy for parallel SAT solvers, in: Theory and Applications of Satisfiability Testing (SAT), 2014, pp. 197–205.

[34] A. Biere, Yet another local search solver and Lingeling and friends entering the SAT Competition 2014, in: A. Belov, M. J. H. Heule, M. Järvisalo (Eds.), Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions, Vol. B-2014-2 of Department of Computer Science Series of Publications B, University of Helsinki, 2014, pp. 39–40.

[35] M. Heule, O. Kullmann, S. Wieringa, A. Biere, Cube and conquer: Guiding CDCL SAT solvers by lookaheads, in: Haifa Verification Conference (HVC), 2011, pp. 50–65.

[36] P. Van Der Tak, M. J. Heule, A. Biere, Concurrent cube-and-conquer, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2012, pp. 475–476.

[37] M. Luby, A. Sinclair, D. Zuckerman, Optimal speedup of las vegas algorithms, Information Processing Letters 47 (4) (1993) 173–180.

[38] J. Ezick, J. Springer, T. Henretty, C. Oh, Extreme sat-based constraint solving with r-solve, in: IEEE Conference on High Performance Extreme Computing (HPEC), IEEE, September, 2014, pp. 1–6.

[39] J. Myers, A. Well, R. Lorch, Research Design and Statistical Analysis: Third Edition, Taylor & Francis, 2013.