Behind the Scene of Solvers Competitions: the *evaluation* Experience

Olivier Roussel
Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens
roussel@cril.univ-artois.fr

Abstract. In principle, running a competition of solvers is easy: one just needs to collect solvers and benchmarks, run the experiments and publish the results. In practice, there are a number of issues that must be dealt with. This paper attempts to summarize the experience accumulated during the organization of several competitions: pseudo-Boolean, SAT and CSP/MaxCSP competitions.

1 Introduction

Many different kinds of solvers competitions are organized nowadays. The main goal of these competitions is to evaluate solvers in the same experimental conditions. Another goal is to help collecting publicly available benchmarks and also help identifying new solvers on the market. In fact, the actual goal of a competition is to help the community identify good ideas implemented in the solvers as well as strange results which may lead to new ideas.

The most visible result of a competition is a ranking of solvers, which is certainly a good motivation to improve one's solver. However, a ranking is necessarily an over-simplified view of a competition. Indeed, there are several ways to look at the solvers results, and determining which solver is the best one is indeed a multi-objective optimization problem, with an additional complication being that users do not agree on the importance of each criteria. Besides, there are necessarily a number of biases which can influence the rankings such as the selection of benchmarks and the experimental conditions (hardware characteristics, time limits, ...). To sum up, it should not be forgotten that the results of a competition are not an absolute truth, but are just a way to collect data about solvers on a large scale and allow anyone to start his own analysis.

The author's experience with organizing competitions started in 2005 with the pseudo-Boolean (PB) track of the SAT 2005 competition [1], co-organized with Vasco Manquinho. At that time, most of the infrastructure was borrowed from the SAT environment previously developed by Laurent Simon and Daniel Le Berre. The first innovation in this track was the introduction of the *runsolver* tool [2] to improve the timing of solvers and to control more precisely the allocated resources. In 2006, the pseudo-Boolean evaluation became independent, and the competition framework was improved into a system called *evaluation*.

This system was used to organize a large number of competitions since then: pseudo-Boolean competitions (PB05, PB06, PB07, PB09, PB10, PB11, PB12) [1], SAT competitions (SAT07, SAT09, SAT11) and CSP/MaxCSP competitions (CPAI06, CPAI08, CSC09) [3]. *evaluation* is also used by local users to perform their own experiments and was extended to support QBF, Max-SAT, AIG, MUS and WCSP solvers.

Each competition introduced a new challenge and the number of solver types that *evaluation* supports today makes it a very versatile system. This paper attempts to summarize the most important features of *evaluation*, in the hope that this can benefit everyone.

2 System requirements

The first reason for developing *evaluation* was the organization of the pseudo-Boolean competitions. Since the beginning, the pseudo-Boolean competition (PB competition for short) had two tracks: one for the decision problem and another one for the optimization problem. Therefore, beyond the requirements inherent to any competition of a specific kind of solver, the *evaluation* framework was immediately faced to a few more requirements to support different kinds of problems as well as different kinds of solvers. This section lists the main requirements that *evaluation* must fulfill.

Support for partial answers: In the optimization track of the PB competition, the ultimate goal is to find the optimal value of the objective function, and prove that it is actually optimal ("OPT" answer) within the given time limit. However, proving optimality is generally hard and the solver may reach the timeout before it can claim that it has proved optimality. In order to be able to compare solvers which have not been able to give the definitive answer "OPT" within the time limit, it was necessary to be able to interrupt the solvers in a way that allowed them to give the best (partial) answer they had found so far. The implemented solution was to send a SIGTERM signal to the solver to warn it that the time limit was reached, and give the solver a few more seconds to give the best answer it had found that far (in this case, a "SAT" answer indicating that a solution was found, but this solution is not necessarily optimal). Once the grace period is expired, the solver is sent a SIGKILL signal to terminate it. This solution assumes that the solver can intercept the SIGTERM signal, which is easy in most languages. However, this interception is impossible in a few languages such as Java. Fortunately, Java offers a hook to call a function before the termination of the program. Otherwise, solvers must use the time limit parameter provided by the competition environment and make sure that they will terminate gracefully before this limit.

Support for incomplete solvers: The next challenge was to add support for incomplete solvers (local search solvers) in the pseudo-Boolean competition. In the SAT competition (and more generally for a decision problem), support for

incomplete solvers is straightforward because the solver stops as soon as it finds a solution. Therefore, the only difference between a complete and incomplete solvers appears on unsatisfiable instances, where incomplete solvers will reach the time limit without providing an answer[1].

The situation is more complicated in the PB competition (and more generally for an optimization problem) because an incomplete solver will keep searching for a better solution until it reaches the timeout. In order to be able to compare incomplete and complete solvers, it is necessary to get the value of the best solution they found and also, the time used to find this solution. Of course, the competition cannot trust any timing given by the solver. The solution was to require the solver to print a specific line each time it finds a better solution, and let the competition environment timestamp each of these lines. The time required to find the best solution found by the solver (without proving optimality) is called T1 in the *evaluation* framework and allows to compare both complete and incomplete solvers on a common basis.

This approach also allowed to obtain for free a plot of the value of the objective function of the solver current solution, as a function of time. Such a graph gives precious information on the convergence of the solver toward the best solution.

Support for different categories: Basically, a SAT solver is able to solve any kind of CNF. Even if it is specialized for a category of instances such as random ones, it is still able to read instances encoding concrete applications even if in practice, it is unlikely to be able to solve it. In the PB and CSP world for example, the situation is different. Not all PB solvers are able to deal with both decision and optimization problems, with both coefficients which fit in a regular 32 bits integer and coefficients that require arbitrary precision arithmetic, with both linear and non-linear constraints. In the same spirit, CSP solvers are not all able to deal with both extensional constraints, intentional constraints and any kind of global constraint.

Therefore, the environment must support different categories of instances defined upon the characteristics of their constraints. Solvers are registered by their authors in one or more of these categories, which indicates that these solvers are technically able to parse and solve these instances. In the PB competition, there are currently up to 8 categories in each of the 2 tracks.

In the CSP world, dealing with global constraints is a challenge of its own. There are several hundreds of global constraints defined, and many solvers implement only a few of them. It is almost impossible to define in advance categories of instances based on the kind of global constraints they contain: this would require enumerating all subsets of the global constraints appearing in the test set, which would not make sense. The proposed solution here is to define a special "UNSUPPORTED" answer that indicates that a solver has no support for a kind of constraint present in an instance. This allows to identify cluster of

[1] Some incomplete solvers may however answer UNSAT in some cases

solvers which are able to solve the same kinds of constraints and compare them on a common basis.

Verification of answers: One of the most fundamental requirement of a competition framework is to verify the answer given by a solver. Indeed, probably one of the major contribution of a competition is to enhance the global quality of solvers by eliminating incorrect solvers during the competition, which is a strong incentive for writing correct solvers! This verification requires that the solver generates a certificate that is checked by the competition framework.

Ideally, this certificate must be cheap to generate, and cheap to verify. This is generally the case for positive answers of a decision problem. For instance, in most cases, a SAT solver can easily print the model that satisfies the CNF. In contrast, certificates for UNSAT answers are much harder to generate and to check, and is the subject of a specific competition. The only check that is performed for UNSAT answers is to verify that no other solver found a solution. The situation is similar for OPT answers, and the only check performed is to verify that no other solver found a better solution.

The MUS (Minimally Unsatisfiable Subset) competition is a specific case. Certificates are easily generated (they are merely a list of clauses that form a MUS) but harder to verify since it must be checked that the MUS is unsatisfiable and that each proper subset is satisfiable. The *evaluation* framework was extended in 2011 to verify these certificates in a post-processing step.

Data recording: A job is a run of a given solver on one instance. During a competition, it is not uncommon that some jobs do not run correctly, either because of problems with the solver or because of problems in the environment (lack of space on /tmp, interactions with processes left running on the host, ...). It is highly desirable to collect a maximum of information on the jobs in order to be able to analyze what actually happened once a problem is detected. In a sense, we need a kind of flight recorder for solvers.

The *evaluation* system stores a lot of information on the job: the host configuration, the solvers parameters, the instance characteristics and the output of the solver. Besides, *runsolver* regularly saves information on the processes started by the solver. In most cases, this is sufficient to identify problems, in which case jobs just have to be run again. Sometimes, some solvers generate tens of gigabytes of output. To protect itself, the environment must limit the size of the solver output. This is done by *runsolver* which preserves the start and the end of the output. The size limit must be chosen with care because some certificates are huge (e.g. > 16 GB).

Some verifications can be done by the competition organizers, but the ultimate verification can only be done by the solver authors, who are the only ones to know exactly how their solver should behave. This is the reason why all the collected data are made available to the authors before the results of the competition are published, so that they can detect problems that would otherwise remain undetected.

Miscellaneous requirements: A first requirement is that the competition framework should have as little interaction as possible with the solver and especially ensure that nothing slows down the solver. Unfortunately, the only way to ensure this would be to run the solver on a non preemptive operating system, which is incompatible with the normal use of a cluster.

In practice, the solver is run on a traditional Unix system, under the monitoring of *runsolver*. This monitoring process necessarily interferes with the solver (access to main memory and to the processor cache, small consumption of CPU time) but the interference is limited [2] and the benefits of *runsolver* exceed its drawbacks.

Another point is that solvers running on different hosts should not interfere, which may happen when instances are read on a network file system. For this reason, the *evaluation* framework first copies both instances and the solver binaries from the network to a local disk (/tmp) and then starts the solver with every generated data stored on the local disk.

To be sure that the instance is correctly copied to the local disk, and that the correct version of the solver is used, a checksum of each copied file is generated and checked against the fingerprint stored in the database.

At last, the environment allows the solver to report additional information (such as the number of nodes, the number of checks,...) which are recorded by the system.

Parallel solver support: More and more solvers are now designed to use the multicores processors which are available on each machine since several years. The environment must obviously record the wall clock time (WC time) and the CPU time of the solver (see section 4.3) but it must also be able to allocate a subset of the cores to the solver (see section 4.1). When a solver is not allocated the complete set of cores available on a host, the system must ensure that only instances of the same solver will run on that host, in order to at least avoid interferences between two solvers designed by different authors.

3 General architecture

In order to ensure privilege separation, the *evaluation* system is implemented as a client/server system. The server manages the dialog with the database and the log file. It is also in charge of granting a job to each client as well as receiving the job results, checking that the solver answer is consistent with the other answers on the same instance, and recording the results in the database.

The client is in charge of receiving a job from the server, copying the solver and the instance to the local disk (/tmp), construct the solver command line and call *runsolver* to monitor the solver execution. *runsolver* stores its data in files on the local disk. At the end of the solver execution, the client runs a verifier program to check the certificate given by the solver. At last, the client copies the files generated by *runsolver* on the local disk to a central directory shared on the network, and reports the results to the server.

The client is also in charge of allocating the cores to the solver, and ensuring that only instances of the same solver are run in parallel on a node.

The last part of the evaluation system deals with the visualization of the results by the users. This part is currently implemented in PHP. Unfortunately, online generation of pages from the databases is too inefficient (the database is several tens of GB large because it contains the results of many competitions). Therefore, HTML pages are generated in advance and stored in a cache. The drawback of this system is that it limits the number of pages that can be generated, because each of them must be stored on the web server (in compressed form). A new system is planned where the generation of web pages would be done by the browser, which would allow more interaction with the user.

4 Resources Allocation and Limits Enforcement

In this section, we detail the problem of allocating resources to both sequential and parallel solvers in a way that is both efficient and as fair as possible.

4.1 Allocation of Cores

The problem of allocating cores to solvers appears when a cluster of nodes with multicores processors is used, which is always the case with recent hardware. The nodes in the cluster used by *evaluation* have two quad-core processors and 32 GB RAM. Obviously, it is highly desirable to optimize the use of the cluster and run concurrently as many solvers as possible on one host. On the other hand, it is also highly desirable to obtain results that are both reproducible and do not depend on external factors such as the other processes running on the system. Besides, if several solvers are run in parallel, we want to measure times that are almost equivalent to the ones of a solver running alone on the same host. Unfortunately, these objectives are contradictory. As soon as several programs are running in parallel (including the *runsolver* process monitoring the solver), they necessarily compete for access to main memory and more importantly to the various cache levels of the processor.

Some experimentation performed on Minisat, indicated that running 8 sequential solvers in parallel (one core allocated to each solver) induced a 35 % time penalty in average, running 4 sequential solvers concurrently on a node (2 cores allocated per solver) implied a 16% penalty in average and at last running 2 sequential solvers concurrently on a node (4 cores allocated per solver) implied almost no penalty (0.4 %) in comparison of running one single sequential solver per node.

In the SAT 2011 competition, it was decided to run 4 sequential solvers in parallel (2 cores per solver) during phase 1 which is used to select the solvers which can enter the second phase. In phase 2 which is the one actually used for ranking solvers, only 2 sequential solvers were run in parallel on a node (4 cores per solver). Parallel solvers where allocated 4 cores in phase 1 (two solver running in parallel on a node) and 8 cores in phase 2 (only one solver per

node). As explained previously, *evaluation* ensured that only instances of the same solver were run in parallel on a given host.

Clearly, a balance must be found between the precision of the time measurements and the number of solvers run in parallel on the cluster. Given the time constraints and the number of solvers submitted to a competition such as the SAT competition, there is little hope to have enough computing resources to afford running one single solver per node in any case.

4.2 Allocation of Memory

Once the policy for allocating cores to solvers is decided, one must decide of the policy for allocating memory to solvers. The basic policy is to reserve a fraction of RAM to the system and to share equally the rest of memory between the solvers running in parallel. Solvers are not allowed to swap on disk, because this kills the hardware and most importantly gives times which are only representative of the hard disk performances. As an example, solvers in the SAT 2011 competition were allowed to use 31GB divided by the number of concurrent solvers. In phase 1, this amounts to 7.7 GB for sequential solvers and 15.5 GB for parallel solvers. In phase 2, this amounts to 15.5 GB for sequential solvers and 31 GB for parallel solvers. Given policy on core allocation, parallel solvers were allocated twice the memory of a sequential solver!

This looks clearly unfair at first, and actually it is, but on the other hand, parallel solvers necessarily need more memory than sequential solvers. Hence, enforcing the same limit would not be fair either! Clearly, we believe that there is no way to be absolutely fair regarding memory allocation for sequential and parallel solvers. The policy chosen in that competition is not perfect, but can be seen as an indirect way to encourage the development of parallel solvers.

4.3 Allocation of Time

In computer science, there are mainly two distinct notions of time: wall clock time and CPU time. The wall clock time (WC time for short) is the real time that elapses between the start and the end of a computing task. The CPU time is the time during which instructions of the program are executed by a processing unit. On a host with a single processing unit, CPU time and wall clock time are equal as long as the system does not interrupt the program. As soon as a time-sharing system is used on a single processing unit, wall clock time will usually be greater than CPU time, because during some time slices the processor will be allocated to another program. On a host with n processing units, if the program is able to use efficiently each of these units and is not interrupted by the system, the CPU time will be equal to n times the wall clock time. Generally speaking, the CPU time is a good measure of the computing effort, while wall clock time corresponds to the user's perception of the program efficiency.

For sequential solvers, there's a clear agreement that CPU time is the right measure of efficiency since it allows to mostly abstract from the perturbation

caused by the computing environment. For parallel solvers, two different points of view exist.

The first point of view is to consider that only the wall clock time matters, which amounts to considering that CPU resources come for free. This might make sense on a desktop computer where the different cores are idle most of the time. However, this leads to approaches which perform redundant computations, such as some portfolio approaches, and clearly waste computing resources. Besides, the assumption that cores come for free does not make sense in larger environments such as clusters or clouds. Each core must be used efficiently.

The second point of view also consider wall clock time but actually puts the emphasis on CPU time. The motivation is that we expect the parallel solver to distribute the computations equally on the different cores and avoid any redundant computation. Therefore, the CPU time used by a parallel solver should not be significantly greater than the CPU time of a sequential solver and the wall clock time of the parallel solver should tend toward the CPU time of the sequential solver divided by the number of cores. Of course, it is well known that this perfect result cannot be obtained because a parallel solver faces problems that the sequential solver doesn't: synchronization problems, contention on memory access,...

In 2009, there has been strong discussions between the organizers of the SAT competition about which point of view should be taken by the competition. In the end, it was decided to put the emphasis on CPU time in order to encourage the community to use efficiently the available cores and also to be able to compare sequential and parallel solvers on a common basis. This lead to enforcing the same CPU limits for both sequential and parallel solvers. In the end, this was unsatisfactory because it showed only one side of the comparison and completely hid the wall clock time information.

In 2011, it was decided to be more neutral and in fact adopt both point of views. Therefore, two different rankings were set up: a CPU ranking and a WC ranking. The WC ranking is based on wall clock time and was expected to promote solvers which use all available resources to give an answer as quickly as possible. In this ranking, timeout is imposed on the wall clock time. The CPU ranking is based on CPU time and was expected to promote solvers which use resources as efficiently as possible. In this ranking, timeout is imposed on CPU time.

Besides, it was decided to organize only one track mixing both sequential and parallel solvers. Indeed, there's no actual reason to differentiate sequential or parallel solvers. The only thing that matters is their performances, either in CPU time or in WC time. It was expected that parallel solvers would perform better in the WC ranking while sequential solvers would perform better in the CPU ranking. Mixing the two kinds of solvers in a same ranking, either CPU or WC based, allows a mostly fair comparison. If a parallel solver does not perform better than a sequential solver in the WC ranking, there is no point in using it. Conversely, if a sequential solver does not perform better than a parallel solver in the CPU ranking, there is no point in using it.

For a chosen timeout T_o and a number n of available cores, the idea was to have a CPU ranking with a limit on CPU time set to T_o and a WC ranking with a limit on WC time set to T_o. Obviously, it was essential to run one single experiment to get both information. Therefore, sequential solvers were run with a CPU limit of T_o and parallel solvers were run with a CPU limit of $n.T_o$. Since the operating system may suspend the solver execution for some time, we have to select a WC limit which is slightly greater than T_o, otherwise it might be impossible to reach the CPU limit in some cases. Generally speaking, the CPU limit is considered more reliable than the WC limit because it presumably does not depend on the other processes running on the system. A post-processing of the results enforces the same CPU or WC limit to generate the CPU and WC rankings respectively.

Choosing the right WC limit for the ranking is actually extremely difficult. On the one hand, it is clear for sequential solvers that the WC limit should be slightly larger than T_o, let's say $T_o + \epsilon$. The value of ϵ can be chosen relatively large because it is only used to compensate delays that are not caused by the solver. On the other hand, for parallel solvers, it makes more sense to set the WC limit to be equal to T_o. Otherwise, a solver that uses all n cores will hit the CPU limit set to $n.T_o$ after a WC time only slightly greater than T_o, but a solver that leaves some core idle may never hit the CPU limit and only hit the WC limit $T_o + \epsilon$. If ϵ is large, this would imply that inefficient parallel solvers would be granted more WC clock time than efficient parallel solvers. Here, the choice of ϵ compensate delays that are caused in part by the solver itself and therefore should tend to 0.

As an example, in the second phase of the 2011 competition, the experiments were performed with a WC limit set to 5100 s for all solvers. Sequential solvers were allowed to use 5000 s CPU time and parallel solvers on 8 cores had a limit set to 40,000 s CPU time. Results were post-processed to enforce a CPU and WC limit of 5000 s for the CPU ranking and a CPU limit of 40,000 s and a WC limit of 5000 s for the WC ranking.

4.4 Enforcing limits

Once resources are allocated to the solvers, limits on these resources must be enforced. This task is not as obvious as it seems. One important problem is that the most straightforward command for measuring the time of a program (the `time(1)` command) frequently fails for solvers running multiple processes, which occurs as soon as a shell script is used to start the solver. Indeed, this command uses times(2) to display the time statistics of the solver. However, this system call only returns the "resources used by those of its children that *have terminated and have been waited for*". This implies that if, for some reason, the parent process doesn't call wait(2), the resources used by the child will be ignored. This also means that these commands cannot enforce reliable limits for multi-process solvers because the resources used by the child are only reported when it terminates.

runsolver was designed to avoid this trap, as well as some others, and implements several other requirements presented at the beginning of this article. A detailed description of *runsolver* is out of the scope of this paper, but can be found in [2]. We only present here its main characteristics.

runsolver is a Linux specific program and is freely available under the Gnu Public License from `http://www.cril.univ-artois.fr/~roussel/runsolver`. Basically, *runsolver* can be seen as the integration of ulimit(1), time(1) and ps(1) with several improvements. It is called with the command line of the solver to run, and parameters specifying the various limits. Once *runsolver* has launched the solver, it periodically monitors the time and memory consumption of the solver processes by fetching the relevant information from the `/proc` filesystem and summing the resource usages[2]. Sanity checks are performed to identify cases where a parent did not wait for its child. As soon as the solver reaches a resource limit, it is gracefully stopped. Process or thread creation or deletion by the solver are also monitored. Periodically, the list of processes run by the solver is saved in a log file in order to allow a post-mortem analysis of what happened.

Each line printed by the solver can be timestamped to identify how much CPU and wall clock time elapsed since the start of the solver. This is a very convenient feature that allows to learn for example how long the solver took to parse the instance or to learn at what time the solver improved its current solution (for optimization problems).

At last, *runsolver* is able to allocate to a solver a given subset of the host cores (with sched_setaffinity(2)) and is able to deal with solvers that generate a huge amount of output (sometimes several tens of GB) by storing only the start and the end of its output.

Since *runsolver* was designed to avoid requiring any root privilege, it runs as a regular program and slightly compete with the solver for CPU usage and memory access. However, the resources used by *runsolver* are very limited and the perturbation is negligible (see [2]).

5 Rankings

The most visible aspect of a competition is to produce a ranking of solvers, but it should be emphasized that such a ranking can only represent one point of view.

Indeed, there are many different ways to look at solvers, and different users generally have different points of views on the comparison of solvers. One point of view is to consider that the solver able to answer on the greatest number of instances is the best one. This is the point of view adopted in several competitions. It has the advantage to be simple and effective, but of course it is somewhat over-simplified. One may also want to consider the number of solved instances in each family, and prefer solvers which have either a balanced number of solved instances in each family, or inversely prefer solvers which solve the most

[2] Memory of threads of a same process is not added, since they share the same address space.

instances in the family of interest to the user. Some users consider the integration of the solver into a wider system and prefer a fast solver to a solver answering more often but which is slower in average. They may also prefer solvers using in average less memory than competitors with similar results.

Alternatively, one may wish to give an advantage to solvers which use new techniques that are the only ones able to solve some instances. Indeed, the purse scoring [4] integrates this point of view. For each instance, a purse of points is divided between the solvers that gave an answer. When only a few solvers are able to solve an instance, they gain more points. This system has interesting properties, but also some drawbacks (the score of a solver depends on the other solvers for example). Several other scoring methods have been proposed [5,6], each with their own pros and cons.

Several other aspects could or should be taken into account in the rankings. The robustness of a solver, that is, its ability to solve an instance which is close to an instance that it already solves (for example instances obtained by shuffling constraints and variables) is a desirable feature. Determinism, that is, the ability to give the same answer in approximately the same time when the solver is run on the same instance several times, is a feature which is important for the adoption of solvers in industrial applications.

Clearly, there are many criteria to compare solvers and expecting to integrate all these criteria into one single ranking is just an utopia. Therefore, one must accept that a competition ranking is just a way to attract contestants, but that it cannot summarize all the details of the picture taken by the competition.

As a last illustration of this point, let us consider the situation of sequential and parallel solvers. It is clear that sequential solvers must be compared on CPU time. Regarding parallel solvers, WC time is clearly an important parameter. Some users consider that the cores present on their computer come for free and disregard CPU time. In our opinion, this is a mistake. CPU time is a resource of its own, which becomes obvious when the solver is integrated into a larger system. The solution adopted in the SAT 2011 competition is to compare solvers on the two criteria: CPU and WC time, without separating sequential and parallel solvers. This generates a CPU ranking and a WC ranking in which each solver appears. The rationale is that, a parallel solver is of no interest if it does not outperform a sequential solver in WC time, and conversely a sequential solver is of no interest if it does not outperform a parallel solver in CPU time. In practice, it has been actually observed that sequential solvers outperformed some parallel solvers in the WC ranking and that parallel solvers outperformed some sequential solvers in the CPU ranking.

6 Conclusion

This paper presented the principles that governed several competitions (PB, SAT, CSP, MUS,...). It can be seen that several issues must be solved during a competition. Several points are open to discussion. Nevertheless, in the end, organizers must choose their own policy. It should be remembered that a com-

petition does not generate an absolute truth: it merely generates a lot of data that can be analyzed in different ways by the community and that contribute to the improvement of solvers, which is the sole actual goal of a competition.

References

1. Manquinho, V., Roussel, O.: The First Evaluation of Pseudo-Boolean Solvers (PB'05). Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 103–143
2. Roussel, O.: Controlling a Solver Execution: the runsolver Tool. Journal on Satisfiability, Boolean Modeling and Computation(JSAT) **7** (nov 2011) 139–144
3. Lecoutre, C., Roussel, O., Van Dongen, M.: Promoting robust black-box solvers through competitions. Constraints **15**(3) (jul 2010) 317–326
4. Van Gelder, A., Le Berre, D., Biere, A., Kullmann, O., Simon, L.: Purse-based scoring for comparison of exponential-time programs. Poster (2005)
5. Nikolic, M.: Statistical methodology for comparison of sat solvers. In Strichman, O., Szeider, S., eds.: SAT. Volume 6175 of Lecture Notes in Computer Science., Springer (2010) 209–222
6. Van Gelder, A.: Careful ranking of multiple solvers with timeouts and ties. In: Proc. SAT (LNCS 6695), Springer (2011) 317–328