Algorithm Configuration for Portfolio-based Parallel SAT-Solving

Holger Hoos [1] and **Kevin Leyton-Brown** [1] and **Torsten Schaub** [2] and **Marius Schneider** [2]

Abstract. Since 2004, the increases in processing power enabled by Moore's law have been primarily achieved by means of multi-core processor architectures. To make effective use of modern hardware when solving hard computational problems, it is therefore necessary to employ parallel solution strategies. In this work, we demonstrate how effective parallel solvers for SAT, one of the most widely studied NP-complete problems, can be produced automatically from any existing sequential, highly parametric SAT solver. Our approach uses an automatic algorithm configurator to produce a set of configurations to be executed in parallel. Applied to the state-of-the-art SAT solver *Lingeling*, our fully automated procedure produced 4-core solvers with speedups of up to 2.79-fold on a diverse set of instances from the *application* category of the 2003–2011 SAT Competitions. Our best automatically generated parallel portfolio of *Lingeling* configurations outperforms *Plingeling*, the gold medal winner of the application track (wallclock time) of the 2011 SAT Competition, and *ManySAT*, the winner of the special prize for parallel solvers for application instances of the 2009 SAT Competition. We furthermore demonstrate that, when applied to the state-of-the-art multi-threaded SAT and ASP solver *clasp*, our automated approach yields parallelization speedups matching those achieved through the considerable efforts of a human expert with extensive knowledge of the solver.

1 Introduction

Over most of the last decade, additional computational power has come primarily in the form of increased parallelism. As a consequence, effective parallel solvers are increasingly key to solving computationally challenging problems. Unfortunately, the manual construction of parallel solvers is nontrivial, often requiring fundamental redesign of existing, sequential approaches. It is thus very appealing to identify generic methods for the construction of parallel solvers from inherently sequential sources. Indeed, the prospect of a substantial reduction in human development cost means that such approaches can be impactful, even if their results are performance does not reach that of special-purpose parallel designs—just as high-level programming languages are useful, even though compiled software tends to fall short of the performance that can be obtained from expert-level programming in assembly language. One promising approach for parallelizing sequential algorithms is the design of parallel algorithm portfolios [15, 8].

In this work, we study generic methods for generating parallel portfolios from a single, highly parametric sequential solver design for a given problem. As such, it can be understood as an instance of the programming by optimization paradigm [13], providing concrete software tools that leverage algorithm configurators and a user-specified design space to substitute for human development effort. In particular, unlike other approaches (further discussed in Section 2), our methods do not depend on the availability of multiple complementary solver designs. We evaluate our methods in the SAT domain, which we chose because it is widely studied and very relevant to academia and industry. We thus have access to well-known state-of-the-art highly parametric solvers, and are assured that the bar for demonstrating efficacy of parallelization strategies is appropriately high.

We consider two scenarios. In the first, there is no communication between component solvers, and the parallel portfolio can be generated fully automatically from a single, sequential parametric solver. In this case, the design space for a parallel portfolio of size k corresponds to the kth Cartesian power of the design space of the given sequential solver. To evaluate our methods in this setting, we chose *Lingeling*, a prominent, highly parametric state-of-the-art SAT solver underlying the parallel solver that won a gold medal in the application (wall-clock) track of the 2011 SAT Competition.

Our second scenario allows for communication between component solvers in a parallel portfolio. Here, component solvers are copies of a single, parametric sequential solver that communicate through a simple mechanism; for example, in SAT, they might share learned clauses (see, e.g., [10].) The communication mechanism is problem-specific and designed by a human expert, resulting in the same design space as in our first scenario, augmented to further include design choices that span the component solvers (the communication mechanism itself, preprocessing strategies, etc). To evaluate our methods in this setting, we chose to study the state-of-the-art, highly parametric, multi-threaded SAT and ASP solver *clasp*.

The key idea underlying our approach for handling both scenarios lies in the use of automated algorithm configurators, which are now quite mature and have been demonstrated to achieve impressive performance improvements for different solvers on many problems (see, e.g., [18, 1, 28, 21, 16, 17]). The configuration spaces arising in the context considered here are very large and therefore present a considerable challenge even to the best configurators. Therefore, in addition to a rather straightforward approach in which all components of a given parallel portfolio are configured simultaneously, we introduce a greedy approach that adds one component solver at a time. Our results demonstrate that this second approach works particularly well and produces parallel portfolios whose performance on standard 4-core CPUs compares favourably with that of well-known, hand-crafted parallel SAT solvers.

[1] Department of Computer Science, University of British Columbia, Vancouver, BC (Canada), {hoos,kevinlb}@cs.ubc.ca

[2] Institute of Computer Science, University of Potsdam, Germany, {torsten,manju}@cs.uni-potsdam.de

2 Related Work

Well before the advent of the current trend towards multi-core computing, the potential benefits of parallel algorithm portfolios were identified in seminal work by Huberman et al. [15]. Gomes & Selman [8] further investigated conditions under which such portfolios outperform their constituent solvers. Both lines of work considered prominent constraint programming problems (graph colouring and quasigroup completion), but neither presented methods for automatically constructing portfolio solvers. More recently, such methods have been introduced for parallel portfolios in which the allocation of computational resources to algorithms in the portfolio is static [23, 29], as well as in cases where it can change over time [6]. All of these methods build a portfolio from a relatively small candidate set of distinct algorithms. While in principle, these methods could also be applied given a set of algorithms expressed implicitly as the configurations of one parametric solver, in practice, they are useful only when the set of candidates is relatively small. The same limitation applies to existing approaches that combine algorithm selection and scheduling, notably *CPHydra* [22], which also relies on cheaply computable features of the problem instances to be solved and selects multiple solvers to be run in parallel.

In contrast, our work is concerned with building parallel portfolios from very large sets of candidate algorithms which are expressed as parameter settings of high-performance solvers such as *Lingeling* and *clasp*. Our approach critically relies on the availability of an effective algorithm configurator, such as *ParamILS* [19, 18], *GGA* [1], or *SMAC* [17, 20]. It is conceptually related to the *Hydra* and *ISAC* procedures for constructing portfolio-based algorithm selectors [28, 21]. Like these methods, our approach uses an algorithm configurator to determine a set of configurations that complement each other well. Furthermore, like *Hydra*, our GREEDY portfolio construction procedure relies on the idea of determining such configurations one at a time, to achieve a maximum incremental performance improvement in each iteration. However, both *Hydra* and *ISAC* construct per-instance algorithm *selectors*: they do not run multiple solvers in parallel, but instead select a single solver based on instance features. To our knowledge, our work is the first to show how to automatically construct effective parallel portfolios from single, highly parametric solvers.

Another conceptually related approach is *aspeed* [14], which computes (parallel) algorithm schedules by taking advantage of the modeling and solving capacities of Answer Set Programming. Unlike our approach, *aspeed* is based on a diverse set of solvers and does not use an algorithm configurator to optimize its configurations.

Parallel SAT solvers have received increasing attention in recent years. *ManySAT* [10, 11, 9] was one of the first parallel SAT solvers. It is a static portfolio solver that uses clause sharing between its components, each of which is a manually configured, DPLL-type SAT solver based on *MiniSat* [5]. *Plingeling* [3, 4] is based on a similar design; its recent version 587, which won a gold medal in the application track of the 2011 SAT Competition (wrt. wall clock time on SAT+UNSAT instances), shares unit clauses as well as equivalences between its component solvers. Similarly, *CryptoMiniSat* [26], which won silver in the application track of the 2011 SAT Competition, shares unit and binary clauses. *clasp* [7] is a state-of-the-art solver for SAT and ASP [2] that supports parallel multithreading (since version 2.0.0) for search space splitting and/or competing strategies, both combinable with a portfolio approach. *clasp* shares unary, binary and ternary clauses, and (optionally) offers a parameterized mechanism for distributing and integrating (longer) clauses. Finally, *ppfolio* [24] is a simple, static parallel portfolio solver for SAT without clause sharing that uses *CryptoMiniSat*, *Lingeling*, *clasp*, *TNM* [27] and *march_hi* [12] in their default configurations as component solvers and won numerous medals in the 2011 SAT Competition. Like the previously mentioned portfolio solvers for SAT, *ppfolio* was constructed manually, but uses a very diverse set of high-performance solvers as its components. Overall, our approach can be understood as an automatic method for replicating the (hand-tuned) success of solvers like *ManySAT*, *Plingeling*, *CryptoMiniSat* or *clasp*, which are inherently based on different configurations of a single parametric solver.

Algorithm 1: Portfolio Configuration Procedure GLOBAL

Input : parametric solver A with configuration space C; configuration space C_g for communication mechanism between component solvers; desired number k of component solvers; instance set I; performance metric m; configurator AC; number n of independent configurator runs; total configuration time t

Output : parallel portfolio solver A^k

for $j := 1..n$ **do**
    obtain configuration c_j by running AC on A^k with configuration space $C^k \times C_g$ on I using m for time t/n
choose $\hat{c} \in \{c_1, \ldots, c_n\}$ for which A^k gives optimal performance on I according to m **return** $\hat{c}$

3 Parallel Portfolio Configuration

We now describe two new methods for generating a parallel solver portfolio from a single parametric solver, A, with configuration space C. We call the given set of problem instances I; our goal is to obtain optimized performance according to a given metric m. (In our experiments, we minimize PAR10 [18].) We use $A^k = [A_1, \ldots, A_k]$ to denote a parallel portfolio with k component solvers, each of which is a configuration of A. The configuration space of A^k is $C^k = \prod_{i=1}^{k} C$ in the case where there is no communication between the component solvers (apart from coordinated launch and termination), and $C^k \times C_g$ in the case where $A_1, \ldots, A_k$ share information throughout a run, where C_g is the set of all possible settings of the parameters of the communication mechanism and any other global logic. Let AC denote a generic algorithm configuration procedure (in our experiments, we used *ParamILS* [19, 18]). Following our standard practice (see e.g., [20]) we perform multiple independent runs of AC, keeping the configuration with the best performance on I. We model the case of non-communicating component solvers as $C_g := \{\emptyset\}$.

Simultaneous configuration of all component solvers (GLOBAL). Our first portfolio configuration method is the straightforward extension of standard algorithm configuration to the construction of a parallel portfolio (see Algorithm 1). Specifically, if A has ℓ parameters, we treat the portfolio A^k as a single algorithm with ℓ^k parameters and configure it directly. As noted above, we perform n parallel runs of AC. These runs can be performed in parallel, meaning that this procedure requires wall clock time of t/n if n cores are available. Nevertheless, the practicality of this approach is limited by the fact that the global configuration space $C^k \times C_g$ to which AC is applied grows exponentially with k. However, given a powerful configurator, a moderate value of k and a reasonably sized C (and C_g), this simple approach could be quite effective, especially when compared to the manual construction of a parallel portfolio.

Algorithm 2: Portfolio Configuration Procedure GREEDY

Input : parametric solver A with configuration space C; configuration space C_g for communication mechanism between component solvers; desired number k of component solvers; instance set I; performance metric m; configurator AC; number n of independent configurator runs; total configuration time t

Output : parallel portfolio solver A^k

1 A^0 := [empty portfolio]
2 **for** $i := 1..k$ **do**
3   **for** $j := 1..n$ **do**
4   obtain configuration c^i_j by running AC on $A^i := A^{i-1}||A$ with configuration space $\prod_{l=1}^{i-1}\{\hat{c}^l\} \times C \times C_g$ on I using m for time $t/(k \cdot n)$, where $A^{i-1}||A$ denotes the portfolio obtained by extending A^{i-1} with algorithm A
5   let $\hat{c}^i \in \{c^i_1, \ldots, c^i_n\}$ be the configuration for which A^i achieved best performance on I according to m, and let $\hat{c}_i$ be the configuration of the component solver most recently added to A^i
6 **return** $\hat{c}^k$

Iterative configuration of component solvers (GREEDY). For use in what we expect to be the typical case where $C^k \times C_g$ is too large to be effectively searched by AC, we introduce an iterative procedure that adds and configures component solvers one at a time (see Algorithm 2). The key idea is to use AC only to configure the component solver added in the given iteration (and the communication mechanism, as applicable, once there are two or more components), leaving all other components clamped to the configurations that were determined for them in previous iterations. The procedure is greedy in the sense that in each iteration i, it attempts to add a component solver to the given portfolio A^{i-1} in a way that myopically maximizes the performance of the new portfolio A^i (Line 4). Obviously, for $k > 1$, even if we assume that AC finds optimal configurations in each iteration, this greedy procedure is not guaranteed to find a globally optimal configuration of the entire portfolio. However, the configuration tasks in each iteration are much easier than those performed by GLOBAL for even moderately sized portfolio, giving us reason to hope that under realistic conditions, GREEDY might perform better than GLOBAL, especially for large configuration spaces C and C_g, and for comparatively modest time budgets t. Finally, notice that this procedure only runs portfolios of size i in each iteration i; therefore, if there is a cost to computing cycles for each parallel CPU or CPU core, there are savings in earlier iterations $i < k$. (However, note that unlike *Hydra*, which GREEDY resembles, we do run entire portfolios in each iteration rather than individual solvers.) Observe that while the sets of n independent configurator runs in Line 4 can be performed in parallel (as in GLOBAL), the choice of the best-performing configuration $\hat{c}^i$ has to be made after each iteration i, introducing a modest overhead compared to the cost of the actual configuration runs.

4 Experiments

To empirically evaluate our approach for creating and optimizing parallel algorithm portfolios, we applied our GLOBAL and GREEDY methods to two state-of-the-art SAT solvers: *Lingeling* and *clasp*. Specifically, we compared performance-optimized sequential and parallel versions of both solvers to our GREEDY method, running on four cores. Finally, we assessed the performance of the parallel solvers obtained using our approach relative to other parallel SAT solvers. A more detailed description of our experimental findings is available at `http://www.cs.uni-potsdam.de/parfolio`.

Scenarios. We compared six experimental scenarios for each solver. We use the terminology *Default-SP* to denote a single-processor solver's default configuration, and analogously *Default-MP4* for an out-of-the-box four-processor version. We contrasted these solver versions with three versions obtained using automated configuration: *Configured-SP*denotes the best (single-processor) configuration obtained from 40 independent configurator runs on a training set, while *Global-MP4* and *Greedy-MP4* represent the portfolios obtained using our methods from Section 3 for $n = 10$ and $k = 4$.

Solvers. We applied our approach to the two highly parameterized, state-of-the-art SAT solvers *Lingeling* version 276 [3] and *clasp* version 2.0.2 [7]. *Lingeling* has 58 parameters, which (after discretization) gave rise to a configuration space of size about 10^{45}. Our parallel portfolio version of *Lingeling* was implemented based on a simple script that runs a given number of *Lingeling* instances independently in parallel and without communication ($C_g := \{\emptyset\}$). We did not apply our methods to *Plingeling*, the 'official' parallel version of *Plingeling*, because it lacks configurable parameters. However, we did compare our methods to *Plingeling*. (Single-processor) *clasp* has 25 parameters, which—discretized by the developer—induce a space of about 10^{13} configurations. *clasp* comes with a native multi-threaded architecture, in which each parallel thread can be configured nearly as flexibly as the sequential solver. Preprocessing is controlled and configured (C_g) globally for all threads. We did not consider active clause sharing in our experiments, but multi-threaded *clasp* passively shares unary, binary, and ternary clauses. Overall, four-threaded *clasp* can be configured in about 10^{53} distinct ways. *clasp*'s default configurations were determined by its main developer with considerable manual effort; the default parallel portfolio version of *clasp*, *Default-MP4*, was entered in the 2011 SAT Competition.

Instance Sets. We conducted our experiments on instances from the *application (industrial)* categories of the 2003–2011 SAT Competitions. Our configuration experiments distinguish a training and a test set. We used the same training set as Schneider and Hoos [25], consisting of 276 instances from the 2003–2009 SAT Competitions. Our test set was comprised of all application (industrial) instances used in the 2003 and 2011 SAT Competitions, with the exception of instances already included in our training set: 679 instances overall. We chose a captime of 600 seconds for solver runs on training instances performed during configuration, and a captime of 5000 seconds (as in the 2011 SAT Competition) when evaluating solvers on the test set.

Evaluation Criteria. All solvers were configured and evaluated based on PAR10 scores [18], which treat timed-out runs as having taken 10 times the captime. We compared solvers using three measures. First, *overall speedup* measures the speedup in terms of total PAR10 scores, disregarding instances from each table in what follows that were not solved by any solver. Second, *(arithmetic) average speedup* takes the average over the set of the compared solvers' speedups, considering only instances that could be solved by both compared solvers. (We note that this measure was previously used both in the 2008 SAT Race and by Hamadi et al. [11]; however, if there

	PAR10	Overall Speedup *vs* Default-SP	Overall Speedup *vs* Configured-SP	Avg. Speedup *vs* Configured-SP	Geo. Avg. Speedup *vs* Configured-SP
Default-SP	3747	1.00	0.93	1.44	0.98
Configured-SP	3499	1.07	1.00	1.00	1.00
Plingeling	3066	1.22	1.14	7.39	1.46
Global-MP4	2734	1.37	1.27	10.47	1.36
Greedy-MP4	1341	2.79	2.61	3.52	1.60

Table 1: PAR10 scores and speedups on application/industrial SAT instances achieved by single-processor (SP) and 4-processor (MP4) versions of *Lingeling*.

	PAR10	Overall Speedup *vs* Default-SP	Overall Speedup *vs* Configured-SP	Avg. Speedup *vs* Configured-SP	Geo. Avg. Speedup *vs* Configured-SP
Default-SP	7560	1.00	0.82	4.46	1.04
Configured-SP	6170	1.23	1.00	1.00	1.00
Default-MP4	2324	3.25	2.65	7.58	2.15
Global-MP4	3604	2.10	1.71	6.36	1.44
Greedy-MP4	2277	3.32	2.71	9.47	2.14

Table 2: PAR10 scores and speedups on application/industrial SAT instances achieved by single-processor (SP) and 4-processor (MP4) versions of *clasp*.

are instances solved by only one solver, disregarding these when measuring speedup can bias results against that solver.) Finally, *geometric average speedup* takes the nth root of the product of the elements of the set of the compared solvers' speedups over the default, again considering only instances that could be solved by both compared solvers.

We now compare the three measures. The overall speedup assesses the speedup obtained in a situation where a stream of problem instances has to be solved, and our test set is representative of that stream. This is the measure we favour, because performance on hard instances is often the most important, because this measure is much less sensitive to outliers, and because it does not require dropping instances that are solved only by the single, best-performing solver. Thus, while we include the other measures in our tables, we do not discuss them in the text in what follows. Average and geometric average speedups are nevertheless also useful for considering situations where there is substantial uncertainty over the difficulty of instances that will ultimately be faced, and therefore consistent speedups across the entire training set (rather than just hard instances in that set) is important. We note that, unlike geometric average speedup, average speedup can give rise to situations where algorithms A and B have speedups > 1 of A against B *and* B against A simultaneously. (To see this, consider running times 1, 2 for A and 2, 1 for B on two given instances.)

We performed all solver and configurator runs on Dell PowerEdge R610 systems with an Intel Xeon E5520 CPU with four cores (2.26GHz), 48GB RAM running 64-bit Scientific Linux.

Configuration Experiments. We used the FocusedILS variant of *ParamILS* (version 2.3.5) [18], one of the best algorithm configurators currently available. To enable fair performance comparisons, in the case of *Configured-SP* and *Global-MP4* we allowed 8 CPU days of configuration time and 1 CPU day for validation runs per configurator run, which amounted to a total of 360 CPU days. (Validation runs were used to choose the best among a set of configurations; they relied on the same training set as the configuration runs. The test set was used only for evaluating the different methods.) For *Greedy-MP4*, we allowed for 2 CPU days of configuration time and 1 CPU days of validation time per configurator run, which amounted to a total of about 300 CPU days for $k = 4$. When using a cluster of dedicated machines with 4-core CPUs, each of those solver versions could be produced within 9 days of wall-clock time.

	PAR1	PAR10	Timeouts
Virtual Best Solver	1334	10480	138
ppfolio	1646	13310	176
Greedy-MP4 (*Lingeling*)	1717	13712	181
Plingeling (587)	1684	13812	183
Greedy-MP4 (*clasp*)	1856	15310	203
clasp (MT)	1837	15357	204
Plingeling (276)	1850	15437	205
ManySAT(1.1)	1887	16003	213
ManySAT(2.0)	1998	17373	232

Table 3: Comparison of our best parallelization approach, GREEDY, with other parallel SAT solvers from the 2011 SAT Competition in the four-processor setting. (The performance of the Virtual Best Solver is the minimal runtime of each instance given a portfolio of solvers.)

Parallelization speedups. Table 1 presents the results of our experiments with *Lingeling* in the communication-free scenario. We observe that single-processor configuration offered very little benefit here, with only small improvements in PAR10 score. Somewhat better results were obtained for *Plingeling*, but despite access to four cores only achieved an overall speedup of 1.22 as compared to the *Lingeling* default. Our *Global-MP4* method outperformed *Plingeling*, but only slightly, achieving an overall speedup of 1.37 times the default. Our *Greedy-MP4* method made the best use of its four cores, achieving a 2.79-fold speedup (see also Figure 1a). Using a permutation test (10 000 iterations, $p = 0.05$), we confirmed that *Greedy-MP4*'s performance significantly exceeded that of the other methods.

Table 2 summarizes the results of our experiments with *clasp*. Here again we observe small gains from configuring the single-processor solver, and *Greedy-MP4* outperforming *Global-MP4*. Overall, *Greedy-*

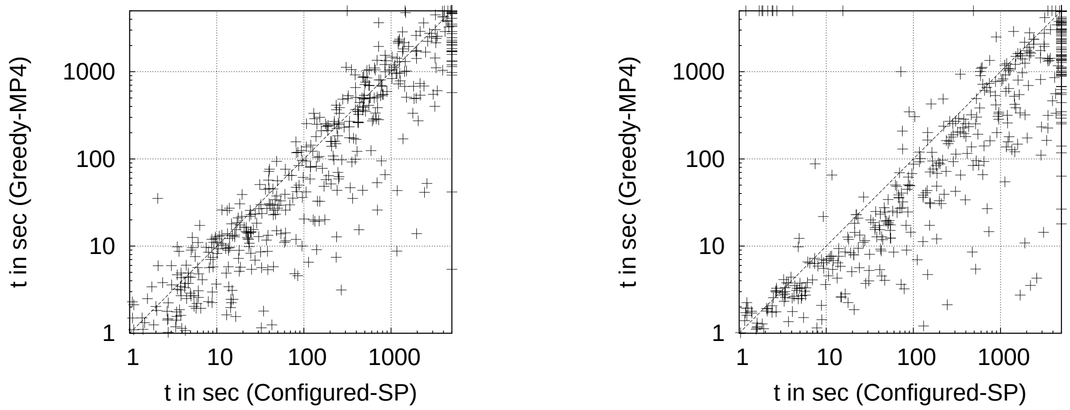

Figure 1: Performance of *Greedy-MP4* vs *Configured-SP* for *Lingeling* (left) and *clasp* (right); each cross represents one SAT instance from our evaluation set.

MP4 performed even better in this domain, achieving a total speedup of 3.32 over the single-processor default (see also Figure 1b). *Greedy-MP4* achieved slightly (but not significantly) better performance as compared to *clasp*'s multi-processor default. However, this default was developed through extensive human effort and (as a SAT Competition entrant) had previously targeted the same data we used to evaluate it. Thus, we see our automated methods' ability to match *Default-MP4*'s performance as an encouraging finding.

Comparison to other parallel solvers. Finally, Table 3 presents a comparison of our methods' performance relative to other 4-processor parallel solvers. We note a few interesting points here. First, *Plingeling*, the 2011 SAT Competition gold medal winner in the application multi-core track, appears only in 3rd place; however, we also note that the competition used 8 processor cores. Second, our *Greedy-MP4* (*Lingeling*), which is based on version 276 of *Lingeling* from 2010, performed as well as the new *Plingeling*, version 587. Third, although the ASP-solver *clasp* was designed for SAT instances more similar to those from the competition's crafted (rather than application) track, *clasp* (in both its default and our *Greedy-MP4* variants) solved more instances than both *ManySAT* versions and slightly more than *Plingeling*, version 276. Fourth, we note that *ManySAT*'s performance was weaker than one might expect given the speedups described in [11]; however, these results were based on (arithmetic average) speedups over the single-processor variant of *ManySAT* rather than *MiniSat* 2.1 (confirmed through personal communication with the authors). Finally, *ppfolio*'s strong performance indicates that portfolios of complementary solvers can yield even stronger performance than parallel portfolios constructed from single parameterized solvers. This is further confirmed by the excellent performance of the perfect per-instance solver selector over the solvers we considered (''virtual best solver''). We aim to consider automatically constructed parallel portfolios that span multiple parametric solvers in future work.

Acknowledgments

T. Schaub and M. Schneider were supported by the DFG projects under SCHA 550/8-1/2 and SCHA 550/9-1.

REFERENCES

[1] C. Ansótegui, M. Sellmann, and K. Tierney, 'A gender-based genetic algorithm for the automatic configuration of algorithms', in *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09)*, volume 5732 of *Lecture Notes in Computer Science*, pp. 142–157. Springer-Verlag, (2009).

[2] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2003.

[3] A. Biere, 'Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010', FMV Reports Series 10/1, Institute for Formal Models and Verification. Johannes Kepler University, (2010).

[4] A. Biere, 'Lingeling and friends at the SAT competition 2011', Technical Report FMV 11/1, Institute for Formal Models and Verification, Johannes Kepler University, (2011).

[5] N. Eén and N. Sörensson, 'An extensible SAT-solver', in *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer-Verlag, (2004).

[6] M. Gagliolo and J. Schmidhuber, 'Learning dynamic algorithm portfolios', *Annals of Mathematics and Artificial Intelligence*, **47**(3-4), pp. 295–328, (2006).

[7] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, 'Conflict-driven answer set solving', in *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pp. 386–392. AAAI Press/The MIT Press, (2007).

[8] C. Gomes and B. Selman, 'Algorithm portfolios', *Artificial Intelligence*, **126**(1-2), pp. 43–62, (2001).

[9] L. Guo, Y. Hamadi, S. Jabbour, and L. Sais, 'Diversification and intensification in parallel SAT solving', in *Proceedings of the Sixteenth International Conference on Principles and Practice of Constraint Programming (CP'10)*, volume 6308 of *Lecture Notes in Computer Science*, pp. 252–265. Springer-Verlag, (2010).

[10] Y. Hamadi, S. Jabbour, and L. Sais, 'Control-based clause sharing in parallel SAT solving', in *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, pp. 499–504. AAAI Press/The MIT Press, (2009).

[11] Y. Hamadi, S. Jabbour, and L. Sais, 'ManySAT: a parallel SAT solver', *Journal on Satisfiability, Boolean Modeling and Computation*, **6**, pp. 245–262, (2009).

[12] M. Heule, M. Dufour, J. van Zwieten, and H. van Maaren, 'March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver', in *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *Lecture Notes in Computer Science*, pp. 345–359. Springer-Verlag, (2004).

[13] H. Hoos, 'Programming by optimisation', *Communications of the ACM*, **55**, pp. 70–80, (2012).

[14] H. Hoos, R. Kaminski, T. Schaub, and M. Schneider, 'aspeed: ASP-based solver scheduling', in *Technical Communications of the Twenty-eight*

International Conference on Logic Programming (ICLP'12). Leibniz International Proceedings in Informatics (LIPIcs), (2012). To appear.

[15] B. Huberman, R. Lukose, and T. Hogg, 'An economic approach to hard computational problems', *Science*, **275**, pp. 51–54, (1997).

[16] F. Hutter, H. Hoos, and K. Leyton-Brown, 'Automated configuration of mixed integer programming solvers', in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'10)*, volume 6140 of *Lecture Notes in Computer Science*, pp. 186–202. Springer, (2010).

[17] F. Hutter, H. Hoos, and K. Leyton-Brown, 'Sequential model-based optimization for general algorithm configuration', in *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION'11)*, volume 6683 of *Lecture Notes in Computer Science*, pp. 507–523. Springer-Verlag, (2011).

[18] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle, 'ParamILS: An automatic algorithm configuration framework', *Journal of Artificial Intelligence Research*, **36**, pp. 267–306, (2009).

[19] F. Hutter, H. Hoos, and T. Stützle, 'Automatic algorithm configuration based on local search', in *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*, pp. 1152–1157, AAAI Press, (2007).

[20] F. Hutter, H. Hoos, and K. Leyton-Brown, 'Parallel algorithm configuration', in *Proceedings of the Sixth International Conference on Learning and Intelligent Optimization (LION'12)*, *Lecture Notes in Computer Science*. Springer-Verlag, (2012). To appear.

[21] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, 'ISAC – instance-specific algorithm configuration', in *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*, pp. 751–756. IOS Press, (2010).

[22] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan, 'Using case-based reasoning in an algorithm portfolio for constraint solving', in *Proceedings of the Nineteenth Irish Conference on Artificial Intelligence and Cognitive Science (AICS'08)*, (2008).

[23] M. Petrik and S. Zilberstein, 'Learning static parallel portfolios of algorithms', in *Proceedings of the International Symposium on Artificial Intelligence and Mathematics (ISAIM 2006)*, (2006).

[24] O. Roussel. Description of ppfolio, 2011. Available at `http://www.cril.univ-artois.fr/~roussel/ppfolio/solver1.pdf`, Last visited on 07-19-2012.

[25] M. Schneider and H. Hoos, 'Quantifying homogeneity of instance sets for algorithm configuration', in *Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12)*, *Lecture Notes in Computer Science*. Springer-Verlag, (2012). To appear.

[26] M. Soos, K. Nohl, and C. Castelluccia, 'Extending SAT solvers to cryptographic problems', in *Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, volume 5584 of *Lecture Notes in Computer Science*, pp. 244–257. Springer-Verlag, (2009).

[27] W. Wei and C. Li. Switching between two adaptive noise mechanism in local search for SAT, 2009. Available at `http://home.mis.u-picardie.fr/~cli/EnglishPage.html`, Last visited on 07-19-2012.

[28] L. Xu, H. Hoos, and K. Leyton-Brown, 'Hydra: Automatically configuring algorithms for portfolio-based selection', in *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, pp. 210–216. AAAI Press, (2010).

[29] X. Yun and S. Epstein, 'Learning algorithm portfolios for parallel execution', in *Proceedings of the Sixth International Conference Learning and Intelligent Optimization (LION'12)*, *Lecture Notes in Computer Science*. Springer-Verlag, (2012). To appear.