clasp: A Conflict-Driven Answer Set Solver

Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub[⋆]

Institut für Informatik, Universität Potsdam,
August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We describe the conflict-driven answer set solver *clasp*, which is based on concepts from constraint processing (CSP) and satisfiability checking (SAT). We detail its system architecture and major features, and provide a systematic empirical evaluation of its features.

1 Introduction

Our new system *clasp* [1] combines the high-level modeling capacities of Answer Set Programming (ASP; [2]) with state-of-the-art techniques from the area of Boolean constraint solving. Unlike existing ASP solvers, *clasp* is originally designed and optimized for conflict-driven ASP solving [3,4], centered around the concept of a *nogood* from the area of constraint processing (CSP). Rather than applying a SAT(isfiability checking) solver to a CNF conversion, *clasp* directly incorporates suitable data structures, particularly fitting backjumping and learning. This includes dedicated treatment of binary and ternary nogoods [5], and watched literals for unit propagation on "long" nogoods [6]. Unlike $smodels_{cc}$ [7], which builds a material implication graph for keeping track of the multitude of inference rules found in ASP solving, *clasp* uses the more economical approach of SAT solvers: For a derived literal, it only stores a pointer to the responsible constraint. Despite its optimized data structures, the implementation of *clasp* provides an elevated degree of abstraction for handling different types of (static and dynamic) nogoods. This paves the way for the future support of language extensions, e.g., aggregates. Different from *smodels* [8] and *dlv* [9], unfounded set detection within *clasp* does not determine greatest unfounded sets. Rather, an identified unfounded atom is immediately falsified, before checking for any further unfounded sets.

We focus on *clasp*'s primary operation mode, viz., conflict-driven nogood learning; its second operation mode runs (systematic) backtracking without learning. Beyond backjumping and learning, *clasp* features a number of related techniques, typically found in SAT solvers based on *Conflict-Driven Clause Learning* (CDCL; [10]). *clasp* incorporates restarts, deletion of recorded conflict and loop nogoods, and decision heuristics favoring literals from conflict nogoods. All these features are configurable via command line options and subject to our experiments. The two major contributions of this paper consist, first, in a detailed description of the system architecture (in Section 2) and, second, in a systematic empirical evaluation of some selected run-time features (in Section 3). Many of these features are based on experiences made in the area of SAT; hence it is interesting to see how their variation affects solving ASP problems.

[⋆] Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

2 System Architecture

The system architecture of *clasp* can be divided into three major components by following the underlying data flow (cf. Figure 1): *clasp* reads ground logic programs in *lparse* format [11], possibly including choice rules, cardinality and weight constraints. The latter constructs are compiled away during *parsing*. The resulting normal rules are then taken by the *program builder* to generate *nogoods* (capturing Clark's completion) and to create an initial positive atom-body-dependency graph (containing only distinct bodies). While all vertices of this graph are associated with assignable variables in the *static data*, only the non-trivial strongly connected components of the positive atom-body-dependency graph are kept and used to initialize the *unfounded set checker*. Note that *clasp* uses *hybrid assignments*, treating atoms and bodies equitably as assignable objects.

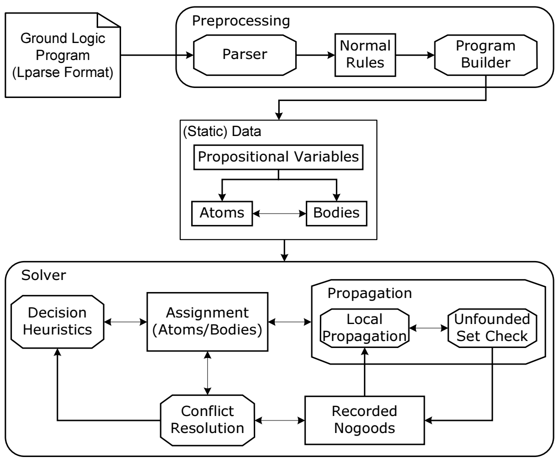

Fig. 1. The system architecture of *clasp*

The elementary data type used in the *solver* is that of a *Boolean constraint* (and thus not restricted to sets of literals). The solver distinguishes static nogoods (see above) that are excluded from nogood deletion and *recorded* nogoods (stemming from conflicts or loops) accumulated during the search. While the former are part of the static data, the latter are kept in a separate database. Also, a learnt nogood maintains an activity counter that is used as a parameter for nogood deletion (see below). Different data structures are used for binary, ternary, and longer nogoods (accounting for the large number of short nogoods capturing Clark's completion). This is complemented by maintaining two *watch lists* [5,6] for each variable, storing all longer nogoods that need to be updated if the variable becomes true or false, respectively.

Variable assignments are either done by *propagation* or via a decision heuristics. *clasp*'s *local* propagation amounts to applying the well-known *unit clause rule* to nogoods (cf. [3]). A variable assigned by local propagation has a pointer to the (unit) nogood it was derived from; this includes unfounded atoms derived from loop nogoods (see [3] for details). During propagation, binary nogoods are preferred over ternary ones, which are preferred over longer nogoods. Also, our propagation procedure is distinct in giving a clear preference to local propagation over *unfounded set* computations. Once an unfounded set U is determined, only a single atom from U is taken to generate a loop nogood that is added to the recorded nogoods. Then, local propagation resumes until a fixed point is reached. This is repeated until there are no non-false atoms left in U. Afterwards, either another unfounded set is found or propagation terminates.

Unfounded set detection within *clasp* combines *source pointers* [12] with the unfounded set computation algorithm in [13]. Notably, it aims at small and "loop-encompassing" rather than greatest unfounded sets, as determined by *smodels* and *dlv*.

Whenever propagation encounters a conflict, *clasp*'s *conflict resolution* is engaged. As described in [3], conflict resolution determines a conflict nogood (that is recorded) and a decision level to jump back to. Backjumping and nogood recording work similar to CDCL with *First-UIP scheme* [10]. The corresponding algorithms are detailed in [3]. For enumerating answer sets, *clasp* uses a novel bounded backjumping approach that is elaborated upon in [4]. Given that *clasp*'s learning and backjumping strategy have already been theoretically as well as experimentally elaborated upon elsewhere [3,4], let us concentrate in what follows on heuristic aspects and in particular investigate how established CSP and SAT strategies apply in the context of ASP. This also provides an overview of the variety of different strategies supported by *clasp*.

clasp's *decision heuristics* depends on whether learning is in effect or not. Without learning, *clasp* relies on *look-ahead* strategies (that extend unit propagation by *failed-literal detection* [14]). When learning, *clasp* uses *look-back* strategies derived from corresponding CDCL-based approaches in SAT, viz., *VSIDS* [6], *BerkMin* [15], and *VMTF* [5]. All of them are conflict-oriented and so primarily influenced by conflict resolution. The heuristic values mainly need to be updated when a new nogood is recorded. Notably, *clasp* leaves it to the user whether this includes loop nogoods or not.

clasp distinguishes two types of *restart policies*. The first starts with an initial number of conflicts after which *clasp* restarts; this threshold is then increased by a factor after each restart. The second policy goes back to Luby et al. [16] and is based on a sequence of numbers of conflicts (e.g., 32 32 64 32 32 64 128 32 $\ldots$ for unit 32) after each of which it restarts. The bounded restart strategy used when enumerating answer sets is described in [4]. Moreover, *clasp* allows for a limited number of *initial randomized runs*, typically with a small restart threshold, in the hope to discover putatively interesting nogoods before actual search starts.

clasp's *nogood deletion* strategy borrows ideas from *minisat* [17] and *berkmin* [15]. It associates an *activity* with each dynamic nogood and limits the number of recorded nogoods by removing nogoods whenever a threshold is reached. The limit is initialized with the size of the input program and increased by a factor every restart. Note that nogood deletion applies to both conflict as well as loop nogoods.

3 Experiments

We conducted experiments on a variety of problem classes. Our comparison includes *clasp* (RC4) in various modes: the normal mode (N) and variants of it changing either the heuristics (H), initial randomized runs (I), restarts (R), or nogood deletion (D):

N The standard mode of *clasp* (RC4) defaults to the following command line options:
  `--heuristic=berkmin` indicates that choices (on atoms and bodies) are done according to an adaption of the *BerkMin* heuristics [15].
  `--lookback-loops=no` indicates that the heuristics ignores loop nogoods.
  `--restarts=simple(100,1.5)` makes *clasp* restart every 100×1.5^k conflicts for $k \geq 0$ (i.e., after 100 150 225 337 506 $\ldots$ conflicts).

--deletion=3,1.1 fixes the size and growth factor of the dynamic nogood database. Initially, *clasp* allows for recording $(|atom(\Pi) \cup body(\Pi)|/3)$ nogoods before nogood deletion is invoked. The size is increased with each restart by the factor `1.1`, given as second parameter.

--loops=common uses a fixed set of bodies when composing the loop nogoods of an unfounded set (alternatives: `distinct` and `shared`; cf. [1]).

H1 **--heuristic=berkmine** modifies the initialization of `berkmin` by counting watched literals rather than taking the original randomized approach.

H2 **--heuristic=vmtfe --lookback-loops=yes** uses an extended adaption of the *VMTF* heuristics [5] and furthermore takes loop nogoods into account. (Actually, the other heuristics could use loop nogoods as well. But only with VMTF, they showed an improvement, while hampering the other heuristics.)

H3 **--heuristic=vsids** uses an adaption of the *VSIDS* heuristics [6].

I **--randomize=50,20** makes *clasp* perform `50` initial runs with a random choice policy before actual search commences; each run is stopped after `20` conflicts.

R1 **--restarts=luby(64)** uses Luby et al.'s restart strategy [16] with base `64`.

R2 **--restarts=simple(16000,1)** is similar to *siege*'s fixed-interval restart strategy [5], cutting of every `16000` conflicts.

R3 **--restarts=simple(700,1)** is similar to *chaff*'s fixed-interval restart strategy [18], cutting of every `700` conflicts.

R4 **--no-restarts** inhibits restarting.

D1 **--deletion=25,1.1** keeps the dynamic nogood database rather small.

D2 **--no-deletion** turns off deletion of dynamic nogoods.

For comparison, we include *smodels* with default settings (S; V2.32) and with its restart option (Sr). We also incorporate *smodels*$_{cc}$ (Scc; V1.08) with option "nolookahead", as recommended by the developers, and *cmodels* (C; V3.65) using *zchaff* (2004.11.15).

All experiments were run on a 800MHz PC on Linux. We report the average time (in seconds) on ten different *shuffles* of an input program. Each run was restricted to 300s time and 512MB RAM. Times exclude parsing, done off-line with *lparse* (V1.0.17). A timeout in *all* 10 runs is indicated by "•"; otherwise, it is taken to be 300s within statistics. The benchmark instances as well as extended results are available at [1,19]. The instances in Table 1 are random programs (1-10); computing bounded spanning trees (11-15), weighted spanning trees (16-20), and Hamiltonian cycles (21-25); game solving for Sokoban (26-35) and Gryzzles (36-40); from bounded model checking (41-52); Social Golfers scheduling (53-57); and machine code superoptimization (58-62). The problems have a variety of different characteristic properties, such as SAT vs UNSAT, random vs structured, tight vs non-tight, etc. Our aim is to give an overview of *clasp*'s performance on a broad palette of problems, from which instances are picked representatively with the only requirement that they are selective.

For brevity, we here only provide a summary of the benchmark results shown in Table 1. For each solver, the last 7 rows show statistics over all runs ($62 \times 10 = 620$ runs per solver). Let us focus on the number of "Timeouts" indicating robustness. (Recall that we shuffled the inputs in order to compensate for luckiness.) It turns out that all different features of *clasp*, that is, heuristics, restarts, and clause deletion, have an impact. Among heuristics, the BerkMin variants N and H1 turned out to be more reliable than VMTF (H2) and VSIDS (H3). Although VMTF is often best, it also leads to

Table 1. Experiments computing *one* answer set

No	Instance	N	H1	H2	H3	I	R1	R2	R3	R4	D1	D2	S	Sr	Scc	C
1	lp.200.00900.9	25.3	24.3	94.2	56.3	25.2	64.4	22.4	55.7	16.1	21	50.6	291.6	291.8	94.6	60.3
2	lp.200.00900.19	25.2	24.6	65.2	44.6	26.7	86.5	16.4	72.9	12.7	23.2	56.6	229.1	231.5	86.9	72.6
3	lp.200.00900.23	20.4	19.8	64	41	22	67.9	16.1	62.1	12.1	21.5	50.5	254.8	244.6	77.2	72.6
4	lp.200.01000.2	21.3	25.3	75.5	47.2	26	60.6	18.9	56.9	13.5	21.7	47	247.1	245.5	70.2	49.8
5	lp.200.01000.22	24.2	21.4	77.1	46.2	25.4	62.1	18.4	55.7	13.3	23.2	43.9	234	232.4	68.8	57.9
6	b5	43.4	45.5	67.6	56.5	45.6	●	158	●	19.4	35.8	216.6	108.5	103.4	●	279.5
7	b9	51	49	72.6	60.6	48.1	●	254.6	●	24.8	37.1	257.4	127.4	131	●	●
8	b10	54.7	52.4	87.8	61.6	55.6	●	243.3	●	25.1	43.4	276.4	165.7	164.9	●	●
9	b17	29.7	25.9	49.7	58	24.6	153.1	50.3	174.1	10.8	21.1	154.6	80.8	76.3	233.1	182.8
10	b26	32.4	36.7	26.7	60.2	55.9	114.7	26.5	151.2	27	17.4	103.8	81.3	177.9	269.9	172.8
11	104_rand_45_250_1727040059_0	48.8	45.9	24.1	52.8	40.6	52.6	45.8	41.7	50.9	48.9	48.9	●	●	296.1	152.3
12	104_rand_45_250_1727040917_0	56.3	51.7	24.9	57.6	39.8	60.7	42.1	42.3	42.2	56.3	56.3	●	●	297.6	184.2
13	104_rand_45_250_1727042043_0	61.7	65	27.5	108.5	40.2	78.8	41.6	56.2	41.6	61.7	61.7	●	●	287.5	102.3
14	104_rand_45_250_1727044175_0	47.4	43.6	24.8	47.8	38.7	53.6	42.3	42.7	42.2	47.5	47.4	●	●	298.7	163.3
15	104_rand_45_250_1727068226_0	48.8	49.1	26.5	57.8	40.7	57	41.2	41.9	41.2	48.8	48.8	●	●	296.9	122.1
16	207_rand_35_138_2077101081_0	12.8	16.3	8.2	18.9	12.9	13.4	11.7	10.9	11.7	12.8	12.8	191.8	225.7	85.8	11.6
17	207_rand_35_138_2077159055_0	10.2	10.6	7.6	13.3	12.8	10.2	10	10.1	10	10.2	10.2	290	296.5	76.7	10.5
18	209_rand_45_138_1119566817_0	23.3	23.1	15.4	31.2	23.4	24.1	25.4	23.6	30.3	23.3	23.3	●	●	215.9	21.4
19	209_rand_45_138_1119569108_0	26.8	27.9	21.4	44.1	24.5	27.7	25.4	25.1	25.5	26.8	26.8	●	●	222.3	19.4
20	209_rand_45_138_1119571853_0	24	25	15.4	37.4	22.5	24.2	22.7	21	22.7	24	24	●	●	202.7	30.4
21	rand_200_1800_1154991214_4	6.6	19.9	28	9.3	10	7.1	24.9	5.2	68.9	6.6	6.6	179.9	71.8	18.7	106.7
22	rand_200_1800_1154991214_7	5.1	18.6	4.1	5.4	9.4	5	27.1	5.4	95.1	5.1	5.1	219.5	123.7	16.8	127.2
23	rand_200_1800_1154991214_9	6	14.9	5.7	5.6	7.9	9.5	60.2	7.9	182	6	6	218.2	57.5	28	120.6
24	rand_200_1800_1154991214_11	5.6	19.7	8	6.9	9	6.5	40	6	93.3	5.6	5.6	●	206.1	17.3	91.9
25	rand_200_1800_1154991214_14	5.9	18.2	6.4	4.9	9	8.3	21.4	5.6	53.1	5.8	5.9	154.6	82.8	21.3	96.5
26	yoshio.2.n16.len15	24	26.5	32.9	56.7	19.3	26.4	26.1	29.4	26.1	32.3	26.1	●	●	63	78.6
27	yoshio.2.n16.len16	24	37.7	48.2	50.6	16.8	34.3	21.9	26.9	19.6	31.2	33.2	●	●	62.2	92.5
28	yoshio.11.n15.len14	4.8	4.9	7.3	28.1	5	4.6	4.7	4.8	4.7	4.9	4.8	●	●	12.8	18.7
29	yoshio.11.n15.len15	4.5	6.2	8.6	31.2	6.3	5.2	5.8	6.2	5.9	5.3	4.5	●	●	12.8	36.4
30	yoshio.36.n14.len13	13.8	13.7	15.7	38.9	9.1	10.4	12.1	12	12	15	43.8	●	●	18.2	22.7
31	yoshio.36.n14.len14	12.3	10.4	12.5	29.2	6.7	9.5	8.9	10.2	8.9	13.3	13.7	●	258.4	21.7	27.7
32	yoshio.46.n13.len12	13.5	15	18.2	24.5	12.9	14.6	13.6	12.7	13.6	17.6	13.4	●	●	34.3	36.3
33	yoshio.46.n13.len13	14.5	20.7	17.7	15.1	11.1	11	15.3	12.9	15.3	11.8	14.5	●	●	48.3	30.7
34	yoshio.52.n12.len11	10.8	13.2	19	23	11.5	12.6	11.3	11.8	11.3	12.7	10.8	180.4	181.5	29.8	32.4
35	yoshio.52.n12.len12	10.3	10.7	13.6	20.5	11.9	9.5	8.9	8.3	8.9	11.6	11.2	●	●	24.4	40.3
36	gryzzles.0	38.2	24.1	44.2	37.5	22.7	12.4	30.4	2.8	210.3	42.3	90.8	●	181	117.1	26.1
37	gryzzles.3	0.3	0.3	0.3	0.3	0.7	0.3	1.2	0.3	1.6	0.2	0.3	21.7	0.5	0.6	1
38	gryzzles.7	0.3	0.5	0.5	0.4	0.8	0.3	3.7	0.4	81.3	0.5	0.3	180.5	3	2	1.1
39	gryzzles.18	0.6	0.7	0.7	0.6	1	0.6	6.3	0.6	34.8	1	0.6	4.8	1.8	2	2.5
40	gryzzles.47	1.3	1.4	1.9	1.7	1.9	1.5	7.6	1.5	116	1.4	1.1	●	18.7	8.5	11.3
41	dp_10.formula1-i-O2-b12	6.9	9.7	14.1	47.5	10.5	11.8	18.3	6.4	15.9	8.7	6.9	165.6	273.3	10.8	38
42	dp_12.formula1-i-O2-b14	44.8	73.6	102.6	200.1	43.6	70.6	77.9	88.1	234.8	38.4	64.6	●	●	75.4	150.5
43	dp_12.formula1-s-O2-b10	3.8	5.6	9.8	10.8	5.4	2.7	3.6	4.3	3.6	3.2	2.9	●	●	6.4	16.6
44	dp_10.fsa-D-i-O2-b10	2.4	0.4	0.3	10.4	4.1	1.5	6.8	0.9	39.7	0.8	3.5	294	6.5	33.9	1.2
45	dp_12.fsa-D-i-O2-b9	●	●	●	●	●	●	●	●	287.6	●	●	●	●	●	●
46	elevator_2-D-i-O2-b12	10.1	8.7	7.8	26.7	10	11.7	8.3	10.2	8.4	10.1	10.1	4.4	22.4	14.4	7.4
47	elevator_4-D-s-O2-b10	3.4	5.2	3.2	7.8	4.5	3.3	4.4	4	4.4	3.5	3.4	97	7.4	21	5.1
48	key_2-D-i-O2-b29	25.2	33.5	39.5	140.2	22.7	31	36.3	30.4	35.1	25.6	25.2	●	●	83.6	50.2
49	key_2-D-s-O2-b29	33.1	32.7	39	91	28	30.6	30.1	30.7	28.7	29.3	35.2	●	●	65	40.8
50	mmgt_3.fsa-D-i-O2-b10	9.6	15.9	23.8	35	6.5	9	10.6	8.7	10.6	8.6	9.6	40.8	16.9	13	9.5
51	mmgt_4.fsa-D-i-O2-b12	180.2	95.4	120.8	217	126.8	76.7	139.5	96	144.6	115.9	180.6	●	274.9	28.3	36.4
52	q_1.fsa-D-i-O2-b17	221.7	187	278.9	293.1	132.3	274.9	176.1	290	161.6	87.8	292.3	●	●	201.1	281.6
53	csp010-SocialGolfer_w3_g3_s6	35	35.7	50.3	103.3	57.9	50.7	34.8	33.4	38.9	84.3	34	●	●	57.2	23.8
54	csp010-SocialGolfer_w4_g3_s6	34.9	41.2	61.7	102.7	60.9	44.3	35.9	38.7	44.2	76	38.9	●	●	64.2	31
55	csp010-SocialGolfer_w6_g3_s4	14.8	24.2	9.5	277.9	97.1	14.8	9.2	20.6	9.3	17.2	16.9	189.2	187.7	40	38.5
56	csp010-SocialGolfer_w6_g3_s5	40.9	60.5	13.7	234.4	141.6	59.3	52.5	49.4	42.9	70.5	55.7	●	●	62.3	40
57	csp010-SocialGolfer_w7_g3_s6	40.1	79.8	18.4	284.5	52.1	91.8	38.7	38.4	45.3	80.6	45.4	●	●	75.1	35.5
58	sequence2-ss2	14.9	15.7	14.7	17.3	14.4	16.1	14.2	14.2	14.2	14.9	14.9	83.5	136.8	115.5	38.2
59	sequence3-ss2	27	27	27.2	27.1	27	26.9	26.9	26.9	26.9	27	26.9	22.8	22.8	24.6	12.8
60	sequence3-ss3	170.7	177	128.5	175.5	128.9	209.8	183.9	141.9	185.7	168.4	170.5	●	●	●	289.5
61	sequence4-ss2	69.2	70.9	80.8	75.8	77.2	73.5	70.1	69.6	70	69.1	69.1	235.6	232.1	225.1	294.8
62	sequence4-ss4	292.5	297.7	293.5	●	●	●	●	●	●	292.7	292.6	●	●	●	●
	Timeouts	23	25	31	66	26	65	33	67	57	22	40	396	342	110	79
	Best	51	56	147	41	64	52	47	64	120	60	42	26	17	21	67
	Worst	25	25	39	77	28	66	34	67	63	22	40	445	384	115	94
	Better	461	379	360	238	431	333	408	378	427	423	386	48	46	103	163
	Worse	159	241	260	382	189	287	212	242	193	197	234	572	574	517	457
	Average	39.9	41.3	45.3	70.5	40.1	61.5	49.4	58.4	53.4	38.5	58.3	233.8	212.7	109	87.2
	Euclidian Distance	341.9	312.2	387.4	685.3	312.6	651.9	476.2	645.9	509	287.2	597.9	1860.5	1753.5	1048.5	825.9

more timeouts. As one might expect, the variant without restarts (R4) is less robust than the restarting variants N and R2. This is also confirmed by *smodels*, where the restart option (Sr) significantly reduces the number of timeouts in comparison to the default setting (S). On the *clasp* variants R1 and R3, we however see that very short restart intervals also degrade performance. Except for *smodels*, all solvers shown in Table 1 use learning and turn out to be more robust than *smodels*. But we also observe that keeping all recorded nogoods, as done by *clasp* variant D2, degrades performance. In contrast, making the dynamic nogood database smaller (D1) was useful on the benchmarked instances. Finally, initial randomized runs (I) tend to slightly increase the solving time when compared to the fastest non-randomized *clasp* variants. However, if the deterministic variants of *clasp* fail, then randomization might be useful. The last 6 rows in Table 1 count how often a solver was "Best", "Worst", and "Better" or "Worse" than the median solving time on a (shuffled) instance, provide its "Average" time over all runs, and finally, the "Euclidian Distance" to the virtual optimum solver (best on all instances) in a 62–dimensional space. Benchmark results for combinations of different options are beyond the scope of this paper. However, the fine-tuning of *clasp* is an ongoing process.

References

1. (`http://www.cs.uni-potsdam.de/clasp`)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
3. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. [20] 386–392.
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. This volume.
5. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. MSc thesis, Simon Fraser University (2004)
6. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC'01. (2001) 530–535
7. Ward, J., Schlipf, J.: Answer set programming with clause learning. In: Proc. LPNMR'04. Springer (2004) 302–313
8. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
9. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL **7**(3) (2006) 499–562
10. Mitchell, D.: A SAT solver primer. Bulletin of the EATCS **85** (2005) 112–133
11. Syrjänen, T.: Lparse 1.0 user's manual. (http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz)
12. Simons, P.: Extending and Implementing the Stable Model Semantics. Dissertation, Helsinki University of Technology (2000)
13. Anger, C., Gebser, M., Schaub, T.: Approaching the core of unfounded sets. In: Proc. NMR'06. Clausthal University of Technology (2006) 58–66
14. Freeman, J.: Improvements to propositional satisfiability search algorithms. PhD thesis, University of Pennsylvania (1995)
15. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT solver. In: Proc. DATE'02. (2002) 142–149
16. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47**(4) (1993) 173–180
17. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT'03. (2003) 502–518
18. Huang, J.: The effect of restarts on the efficiency of clause learning. [20] 2318–2323.
19. (`http://asparagus.cs.uni-potsdam.de`)
20. Proc. IJCAI'07, AAAI Press/The MIT Press (2007).