# JWASP: A New Java-Based ASP Solver

Mario Alviano, Carmine Dodaro, and Francesco Ricca

Department of Mathematics and Computer Science,
University of Calabria, 87036 Rende (CS), Italy
{alviano,dodaro,ricca}@mat.unical.it

**Abstract.** Answer Set Programming (ASP) is a well-known declarative programming language for knowledge representation and non-monotonic reasoning. ASP solvers are usually written in C/C++ with the aim of extremely optimizing their performance. Indeed, C/C++ allow for several low level optimizations, which however come at the price of a less portable implementation. This is a problem for some real world use cases which do not actually require an extremely efficient computation, but would benefit from a platform-independent and easily-deployable implementation. Motivated by such use cases, we develop JWASP, a new ASP solver written in Java and extending the open source library SAT4J in order to process ASP programs with atomic heads. We also report on a preliminary experiment assessing the performance of JWASP, whose results are encouraging: JWASP is a good candidate as an alternative ASP solver for platform-independent applications, which cannot rely on current ASP solvers.

## 1 Introduction

Answer Set Programming (ASP) [5] is a declarative programming paradigm, which has been proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find them [5]. The availability of solvers has made possible the application of ASP for solving complex problems arising in several areas [1, 6], including AI, knowledge representation and reasoning, databases, bioinformatics. Recently ASP has been also used to solve a number of industry-level applications [7, 21].

Answer set programming is computationally hard, and modern ASP solvers are usually based on one of two alternative approaches. The first of these approaches consists in implementing a native algorithm by adapting SAT solving techniques [22]. In particular, CDCL backtracking with learning, restarts, and conflict-driven heuristics is extended with ASP-specific propagation techniques such as support inference via Clark's completion, and well-founded inference via source pointers [23]. The second approach resorts on rewriting techniques into SAT formulas, which are then evaluated by an off the shelf SAT solver [13].

ASP solvers, like SAT solvers, are developed having in mind the (often well-deserved) goal of maximizing performance. For this reason, ASP solvers are usually written in C/C++, a programming language that is suited for implementing several low level optimizations, but at the price of a reduced portability. This is a problem for some real world

use cases which do not actually require the highest available performance in computation, but would benefit from a platform-independent and easily-deployable implementation. For example, the iTravel system [20] takes advantage of some ASP-based web services implemented as Java servlets interacting with DLV [16] via the DLV WRAPPER API [19]. Usually, Java servlets are easily exportable as WAR archives, which are then deployable to different servers by simply copying the archives. Such a simplicity was not possible with the ASP-based web services because different versions of DLV were required for servers running different operating systems. A similar issue also affects the distribution of ASPIDE [9], an IDE for ASP developed in Java which must include different versions of an ASP solver for different operating systems. An ASP solver implemented in Java would simplify the distribution of ASPIDE, not preventing the possibility to run other ASP solvers written in C/C++ if needed.

If on the one hand Java provides all the means for implementing a platform-independent ASP solver, on the other hand the following questions have to be answered: How much overhead is introduced? Is the performance of an ASP solver written in Java acceptable when compared with state of the art ASP solvers? Motivated by the needs arising in different use cases, and in order to answer these two questions, we developed JWASP (`https://github.com/dodaro/jwasp.git`), a new ASP solver written in Java. JWASP is based on the open source library SAT4J [15]. In particular, JWASP extends SAT4J in order to process ASP programs with atomic heads.

A preliminary experiment assessing the performance of JWASP has been conducted on benchmarks from the previous ASP competitions [1, 6]. In particular, JWASP was compared with the following state of the art ASP solvers: the native CLASP 3.1.1 [11] and WASP [3]; the rewriting-based LP2SAT endowed by GLUCOSE [4]; and LP2SAT endowed by SAT4J [15]. The results are encouraging. In fact, even if JWASP cannot match the performance of CLASP, which is actually expected, it can compete with a prominent rewriting-based ASP solver using GLUCOSE. Our experiment highlights that JWASP is a good candidate as an alternative ASP solver for platform-independent applications, where conventional solvers cannot be used or might not be comfortably integrated.

## 2 Preliminaries

Syntax and semantics of propositional logic and propositional ASP are briefly introduced in this section.

### 2.1 Propositional Logic

*Syntax.* Let $\mathcal{A}$ be a fixed, countable set of (Boolean) variables, or (propositional) atoms, including $\bot$. A *literal* $\ell$ is either an atom $a$, or its negation $\neg a$. A *clause* is a set of literals representing a disjunction, and a propositional formula $\varphi$ is a set of clauses representing a conjunction, i.e., only formulas in *conjunctive normal form* (CNF) are considered here.

*Semantics.* An interpretation $I$ is a set of literals over atoms in $\mathcal{A} \setminus \{\bot\}$. Intuitively, literals in $I$ are true, literals whose complement is in $I$ are false and the remaining literals

are undefined. An interpretation $I$ is total if there are no undefined literals, otherwise $I$ is partial. An interpretation $I$ is inconsistent if for an atom $a$ both $a$ and $\neg a$ are in $I$. Relation $\models$ is inductively defined as follows: for $a \in \mathcal{A}$, $I \models a$ if $a \in I$, and $I \models \neg a$ if $\neg a \in I$; for a clause $c$, $I \models c$ if $I \models \ell$ for some $\ell \in c$; for a formula $\varphi$, $I \models \varphi$ if $I \models c$ for all $c \in \varphi$. If $I \models \varphi$ then $I$ is a *model* of $\varphi$, $I$ *satisfies* $\varphi$, and $\varphi$ is true w.r.t. $I$. If $I \not\models \varphi$ then $I$ is not a model of $\varphi$, $I$ *violates* $\varphi$, and $\varphi$ is false w.r.t. $I$. Similar for literals, and clauses. A formula $\varphi$ is *satisfiable* if there is an interpretation $I$ such that $I \models \varphi$; otherwise, $\varphi$ is *unsatisfiable*.

*Example 1.* Consider the following formula $\varphi$:

$$\{a, \neg b\} \qquad \{b, \neg a\} \qquad \{\neg a\} \qquad \{c\} \qquad \{c, \neg b\}$$

$\varphi$ is satisfiable and the interpretation $I = \{\neg a, \neg b, c\}$ is a model. $\lhd$

### 2.2 Answer Set Programming

*Syntax.* Let $\sim$ denote *negation as failure*. A *$\sim$-literal* (or just literal when clear from the context) is either an atom (a positive literal), or an atom preceded by $\sim$ (a negative literal). A logic program $\Pi$ is a finite set of rules of the following form:

$$a \leftarrow b_1, \ldots, b_k, {\sim}b_{k+1}, \ldots, {\sim}b_m \tag{1}$$

where $m \geq 0$, and $a, b_1, \ldots, b_m$ are atoms in $\mathcal{A}$. For a rule $r$ of the form (1), set $\{a\}$ is called *head* of $r$, and denoted $H(r)$; conjunction $b_1, \ldots, b_m, {\sim}b_{k+1}, \ldots, {\sim}b_m$ is named *body* of $r$, and denoted $B(r)$; the sets $\{b_1, \ldots, b_k\}$ and $\{b_{k+1}, \ldots, b_m\}$ of positive and negative literals in $B(r)$ are denoted $B^+(r)$ and $B^-(r)$, respectively. A *constraint* is a rule $r$ such that $H(r) = \{\bot\}$.

*Semantics.* An interpretation $I$ is a set of $\sim$-literals over atoms in $\mathcal{A} \setminus \{\bot\}$. Relation $\models$ is extended as follows: for a negative literal ${\sim}a$, $I \models {\sim}a$ if ${\sim}a \in I$; for a conjunction $\ell_1, \ldots, \ell_n$ ($n \geq 0$) of literals, $I \models \ell_1, \ldots, \ell_n$ if $I \models \ell_i$ for all $i \in [1..n]$; for a rule $r$, $I \models r$ if $H(r) \cap I \neq \emptyset$ whenever $I \models B(r)$; for a program $\Pi$, $I \models \Pi$ if $I \models r$ for all $r \in \Pi$. The definition of a stable model is based on a notion of program reduct [12]: Let $\Pi$ be a normal logic program, and $I$ an interpretation. The reduct of $\Pi$ w.r.t. $I$, denoted $\Pi^I$, is obtained from $\Pi$ by deleting each rule $r$ such that $B^-(r) \cap I \neq \emptyset$, and removing negative literals in the remaining rules. An interpretation $I$ is an answer set for $\Pi$ if $I \models \Pi$ and there is no total interpretation $J$ such that $J \cap \mathcal{A} \subset I \cap \mathcal{A}$ and $J \models \Pi^I$. The set of all answer sets of a program $\Pi$ is denoted $SM(\Pi)$. Program $\Pi$ is *coherent* if $SM(\Pi) \neq \emptyset$; otherwise, $\Pi$ is *incoherent*.

*Example 2.* Consider the following program $\Pi$:

$$\begin{array}{lll} a \leftarrow c & a \leftarrow b, {\sim}e & b \leftarrow a, {\sim}e \\ c \leftarrow {\sim}d & d \leftarrow {\sim}c & e \leftarrow {\sim}d \end{array}$$

$I = \{a, {\sim}b, c, {\sim}d, e\}$ is an answer set of $\Pi$. $\lhd$

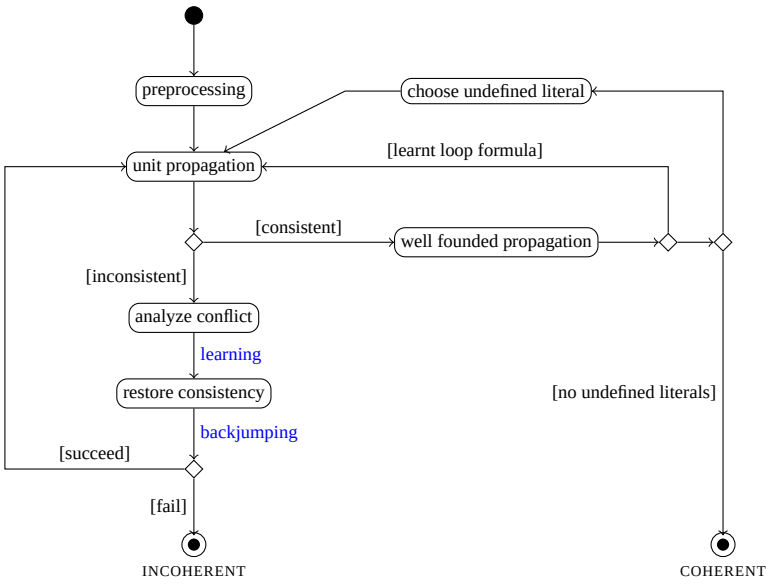


**Fig. 1.** Computation of an answer set in JWASP.

## 3 Answer Set Computation in JWASP

In this section we first review the algorithms implemented in JWASP for the computation of an answer set, and then we describe how these were implemented by extending SAT4J. The presentation is properly simplified to focus on the main principles.

### 3.1 Main Algorithms

The main algorithm is depicted in Figure 1.

*Preprocessing.* The first step is a preprocessing of the input program $\Pi$, that is transformed into a propositional formula called the *Clark's completion* of the program $\Pi$, denoted $Comp(\Pi)$. This step is performed since answer sets are supported models [17]. A model $I$ of a program $\Pi$ is *supported* if each $a \in I \cap \mathcal{A}$ is supported, i.e., there exists a rule $r \in \Pi$ such that $H(r) = a$, and $B(r) \subseteq I$. In more detail, given a rule $r \in \Pi$, let $aux_r$ denote a fresh atom, i.e., an atom not appearing elsewhere, the completion of $\Pi$ consists of the following clauses:

- $\{\neg a, aux_{r_1}, \ldots, aux_{r_n}\}$ for each atom $a$ occurring in $\Pi$, where $r_1, \ldots, r_n$ are the rules of $\Pi$ whose head is $a$;
- $\{H(r), \neg aux_r\}$ and $\{aux_r\} \cup \bigcup_{a \in B^+(r)} \neg a \cup \bigcup_{a \in B^-(r)} a$ for each rule $r \in \Pi$;
- $\{\neg aux_r, \ell\}$ for each $r \in \Pi$ and $\ell \in B(r)$.

After computing the Clark's completion $Comp(\Pi)$, the input is further simplified applying classical preprocessing techniques of SAT solvers [8], and then the nondeterministic search takes place.

*CDCL Algorithm.* The main ASP solving algorithm is similar to the CDCL procedure of SAT solvers. In the beginning a partial interpretation $I$ is set to $\emptyset$. Function unit propagation extends $I$ with those literals that can be deterministically inferred. This function returns false if an inconsistency (or conflict) is detected, true otherwise. When an inconsistency is detected, the algorithm analyzes the inconsistent interpretation and learns a clause using the *1-UIP* learning scheme [18]. The learned clause models the inconsistency in order to avoid exploring the same search branch several times. Then, the algorithm unrolls choices until consistency of $I$ is restored, and the computation resumes by propagating the consequences of the clause learned by the conflict analysis. If the consistency cannot be restored, the algorithm terminates returning INCOHERENT. When no inconsistency is detected, the well founded propagation (detailed in the following) checks whether $I$ is *unfounded-free*. In case $I$ is not unfounded-free a clause is added to $Comp(\Pi)$ and unit propagation is invoked. If $I$ is unfounded-free and the interpretation $I$ is total then the algorithm terminates returning COHERENT and $I$ is an answer set of $\Pi$. Otherwise, an undefined literal, say $\ell$, is chosen according to some heuristic criterion. The computation then proceeds on $I \cup \{\ell\}$. Unit propagation and well founded propagation are described in more detail in the following.

*Propagation rules.* JWASP implements two deterministic inference rules for pruning the search space during answer set computation. These propagation rules are named *unit* and *well founded*. Unit propagation is applied first. It returns false if an inconsistency arises. Otherwise, well founded propagation is applied. Well founded propagation may learn an implicit clause in $\Pi$, in which case unit propagation is applied on the new clause. More in details, unit propagation is as in SAT solvers: An undefined literal $\ell$ is inferred by unit propagation if there is a clause $c$ that can be satisfied only by $\ell$, i.e., $c$ is such that $\ell \in c$ is undefined and all literals in $c \setminus \{\ell\}$ are false w.r.t. $I$. Concerning well founded propagation, we must first introduce the notion of an unfounded set. A set $X$ of atoms is *unfounded* if for each rule $r$ such that $H(r) \cap X \neq \emptyset$, at least one of the following conditions is satisfied: (i) a literal $\ell \in B(r)$ is false w.r.t. $I$; (ii) $B^+(r) \cap X \neq \emptyset$. Intuitively, atoms in $X$ can have support only by themselves. Well founded propagation checks whether the interpretation contains an unfounded set $X$. In this case, it learns a clause forcing falsity of an atom in $X$. Clauses for other atoms in $X$ will be learned on subsequent calls to the function, unless an inconsistency arises during unit propagation. In case of inconsistencies, indeed, the unfounded set $X$ is recomputed.

### 3.2 Implementation

The implementation of a modern and efficient ASP solver requires the implementation of at least three modules. The first module is the parser of a ground ASP program. The second module computes the Clark's completion. The third module implements the CDCL backtracking algorithm extended by applying well founded propagation as presented in Section 3.1. Concerning the parser, JWASP accepts as input normal ground

programs expressed in the numeric format of GRINGO [10]. The Clark's completion is computed after the whole program has been parsed. The third module is implemented by JWASP exploiting the open source Java library SAT4J [15]. In particular, SAT4J provides an implementation of the base CDCL algorithm. JWASP extends this algorithm by modifying the propagate function of SAT4J, which in our solver includes the well founded inference rule. In particular, specific data structures and the algorithm for computing unfounded sets are introduced in JWASP which are not provided by SAT4J.

## 4 Experiment

The performance of JWASP was compared with CLASP 3.1.1 and LP2SAT [13]. CLASP is a native state of the art ASP solver, while LP2SAT is an ASP solver based on a rewriting of the ASP program into a SAT formula that is evaluated using a SAT solver. Two variants of LP2SAT were considered, namely LP2GLUCOSE and LP2SAT4J, which use GLUCOSE [4] and SAT4J [15] as SAT solver, respectively. All the ASP solvers use GRINGO [10] as grounder. The experiment concerns a comparison of the solvers on publicly available benchmarks used in the 3rd and 4th ASP competitions [1, 6]. The experiment was run on a four core Intel Xeon CPU X3430 2.4 GHz, with 16 GB of physical RAM, and operating system Debian Linux. Time and memory limits were set to 600 seconds and 15 GB, respectively. Performance was measured using the tools pyrunlim and pyrunner (`https://github.com/alviano/python`).

Table 1 summarizes the number of solved instances and the average running time in seconds for each solver. In particular, the first column reports the considered benchmarks; the remaining columns report the number of solved instances within the time-out (solved), and the running time averaged over solved instances (time). The first observation is that JWASP outperforms the rewriting-based LP2SAT4J. In fact, JWASP solved 17 more instances than LP2SAT4J and it is in general faster. The advantage of JWASP is obtained in 3 different benchmarks, namely KnightTour, MazeGeneration, and Numberlink, where JWASP solves 5, 7, and 3 more instances than LP2SAT4J. Once the SAT solver backhand is replaced by GLUCOSE, a clear improvement of performance is measured. LP2GLUCOSE is clearly faster (it solves 20 instances more) than LP2SAT4J. In

**Table 1.** Solved instances and average running time.

| | | LP2SAT4J | | JWASP | | LP2GLUCOSE | | WASP | | CLASP | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Track** | **#** | **sol.** | **avg t** | **sol.** | **avg t** | **sol.** | **avg t** | **sol.** | **avg t** | **sol.** | **avg t** |
| GraphColouring | 30 | 8 | 47.45 | 7 | 31.07 | 14 | 124.02 | 8 | 66.15 | 13 | 129.98 |
| HanoiTower | 30 | 27 | 120.80 | 26 | 166.57 | 30 | 10.41 | 30 | 33.83 | 28 | 53.18 |
| KnightTour | 10 | 2 | 67.66 | 7 | 52.03 | 3 | 24.37 | 8 | 4.39 | 10 | 57.95 |
| Labyrinth | 30 | 14 | 222.34 | 17 | 158.44 | 18 | 151.70 | 26 | 72.64 | 26 | 48.05 |
| MazeGeneration | 10 | 3 | 332.46 | 10 | 5.06 | 4 | 164.15 | 10 | 3.10 | 10 | 1.04 |
| Numberlink | 10 | 4 | 98.05 | 7 | 7.67 | 5 | 164.67 | 8 | 12.71 | 8 | 7.91 |
| SokobanDecision | 10 | 6 | 46.57 | 7 | 61.42 | 10 | 59.34 | 9 | 92.15 | 10 | 42.91 |
| **Total** | 130 | 64 | 133.72 | 81 | 100.50 | 84 | 82.45 | 99 | 44.75 | 105 | 52.48 |

this case, since the rewriting technique is the same, the difference of performance is due to the fact that GLUCOSE outperforms LP2SAT4J. The performance gap between C++ and Java implementations can be observed also by comparing WASP and JWASP. In particular, WASP solves 18 more instances than JWASP. The differences are noticeable in Labyrinth where WASP solves 9 more instances than JWASP. Similar considerations hold by comparing CLASP and JWASP. In fact, the former is in general faster solving 24 more instances than the latter. Finally, it is important to note that JWASP is basically comparable in performance with LP2GLUCOSE (the latter solves only 3 instances more than the former). An in-depth analysis shows that JWASP is faster in KnightTour and MazeGeneration solving 4 and 6 instances more than LP2GLUCOSE, respectively. On the contrary, LP2GLUCOSE is faster than JWASP in GraphColouring, HanoiTower, and SokobanDecision. We observe that the main advantage of JWASP over LP2GLUCOSE is registered (as expected) in the benchmarks in which the well founded propagation (implemented natively by JWASP) is applied, such as KnightTour and MazeGeneration.

## 5 Discussion

During recent years, ASP has obtained growing interest since efficient implementations were available. For reason of efficiency, most of the modern ASP solver are implemented in C++. To the best of our knowledge, the only previous Java-based ASP solver was JSMODELS [14], which is not developed anymore. JSMODELS was based on SMODELS featuring the DPLL algorithm and lookahead heuristics. From an abstract point of view, JWASP is more similar to modern ASP solvers, like WASP [2, 3] and CLASP [11]. In fact, all the three solvers are based on CDCL algorithm and source pointers for the computation of unfounded sets. However, JWASP is implemented in Java and thus it is a cross-platform and more portable implementation. An alternative to the development of a native solver is to rewrite the input program into a CNF formula, as done by the family of solvers LP2SAT [13]. This alternative approach can be applied to obtain a Java-based solver by endowing LP2SAT with a Java-based SAT solver such as SAT4J. This approach is less efficient than JWASP in the experimental analysis reported in this paper. It is worth noting that, both JWASP and LP2SAT apply the Clark's completion [17]. Thus, the main difference between JWASP and LP2SAT4J consists of the native computation of unfounded set of JWASP, which is obtained by using an algorithm based on source pointers introduced by SMODELS [23].

In this paper we reported on the new Java-based ASP solver JWASP built on the top of the SAT solver SAT4J. The new solver was compared with both C++ and Java-based approaches. In our experiment, JWASP outperforms the Java-based alternative LP2SAT4J, and it is competitive with LP2GLUCOSE. However, as expected, JWASP is in general slower than the native solvers. This confirms that C++ implementations are usually much faster than Java-based approaches as also noted in [15]. Future work concerns the extension of JWASP for handling optimization constructs and cautious reasoning.

## References

1. M. Alviano, F. Calimeri, G. Charwat, M. Dao-Tran, C. Dodaro, G. Ianni, T. Krennwallner, M. Kronegger, J. Oetsch, A. Pfandler, J. Pührer, C. Redl, F. Ricca, P. Schneider, M. Schwen-

gerer, L. K. Spendier, J. P. Wallner, and G. Xiao. The fourth answer set programming competition: Preliminary report. In P. Cabalar and T. C. Son, editors, *LPNMR*, LNCS, pages 42–53, 2013.
2. M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca. WASP: A native ASP solver based on constraint learning. In P. Cabalar and T. C. Son, editors, *LPNMR*, volume 8148 of *LNCS*, pages 54–66. Springer, 2013.
3. M. Alviano, C. Dodaro, N. Leone, and F. Ricca. Advances in WASP. In F. Calimeri, G. Ianni, and M. Truszczynski, editors, *LPNMR*, volume 9345 of *LNAI*, page (to appear), 2015.
4. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In C. Boutilier, editor, *IJCAI*, pages 399–404, 2009.
5. G. Brewka, T. Eiter, and M. Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
6. F. Calimeri, G. Ianni, and F. Ricca. The third open answer set programming competition. *TPLP*, 14(1):117–135, 2014.
7. C. Dodaro, B. Nardi, N. Leone, and F. Ricca. Allotment problem in travel industry: A solution based on ASP. In B. ten Cate and A. Mileo, editors, *RR*, volume 9209 of *LNCS*. Springer, 2015.
8. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
9. O. Febbraro, K. Reale, and F. Ricca. ASPIDE: integrated development environment for answer set programming. In J. P. Delgrande and W. Faber, editors, *LPNMR*, volume 6645 of *LNCS*, pages 317–330. Springer, 2011.
10. M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In J. P. Delgrande and W. Faber, editors, *LPNMR*, volume 6645 of *LNCS*, pages 345–351. Springer, 2011.
11. M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.
12. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.
13. T. Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.
14. H. V. Le and E. Pontelli. A Java Based Solver for Answer Set Programming.
15. D. Le Berre and A. Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.
16. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM TOCL*, 7(3):499–562, 2006.
17. F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.
18. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.
19. F. Ricca. The DLV java wrapper. In F. Buccafurri, editor, *AGP*, pages 263–274, 2003.
20. F. Ricca, A. Dimasi, G. Grasso, S. M. Ielpa, S. Iiritano, M. Manna, and N. Leone. A logic-based system for e-tourism. *Fundam. Inform.*, 105(1-2):35–55, 2010.
21. F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano, and N. Leone. Team-building with answer set programming in the gioia-tauro seaport. *TPLP*, 12(3):361–381, 2012.
22. J. P. M. Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
23. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.