To appear in the *Journal of Experimental & Theoretical Artificial Intelligence*
Vol. 00, No. 00, Month 20XX, 1–24

Effectively Solving NP-SPEC Encodings by Translation to ASP

Mario Alviano[a] and Wolfgang Faber[a,b]

[a]*Department of Mathematics and Computer Science, University of Calabria, Italy*
[b]*School of Computing and Engineering, University of Huddersfield, UK*
Email: mario@alviano.net, wf@wfaber.com

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

NP-SPEC is a language for specifying problems in NP in a declarative way. Despite the fact that the semantics of the language was given by referring to Datalog with circumscription, which is very close to ASP, so far the only existing implementations are by means of ECL^iPS^e Prolog and via Boolean satisfiability solvers. In this paper, we present translations from NP-SPEC into ASP, and provide an experimental evaluation of existing implementations and the proposed translations to ASP using various ASP solvers. The results show that translating to ASP clearly has an edge over the existing translation into SAT, which involves an intrinsic grounding process. We also argue that it might be useful to incorporate certain language constructs of NP-SPEC into mainstream ASP.

1. Introduction

NP-SPEC is a language that was proposed in (Cadoli, Ianni, Palopoli, Schaerf, & Vasile, 2000; Cadoli, Palopoli, Schaerf, & Vasile, 1999) in order to specify problems in the complexity class NP in a simple, clear, and declarative way. The language is based on Datalog with circumscription, in which some predicates are circumscribed, while others are not and are thus "left open". In particular, the idea at the basis of NP-SPEC is to provide a few constructs, called *metafacts*, for specifying the search space of an NP problem, that is, the relations to guess in order to solve instances of the problem. The simplest metafact in NP-SPEC is used to guess a subset of a relation, but more sophisticated metafacts are also available to guess, for example, a permutation or a partitioning of the extension of a predicate. Actually, the semantics of these practical constructs is defined by means of reductions to the metafact for guessing a subset of a relation.

The original software system supporting NP-SPEC was described in (Cadoli et al., 2000) and was written in the ECL^iPS^e Constraint Programming System, based on Prolog. A second software system, SPEC2SAT[1], was proposed in (Cadoli, Mancini, & Patrizi, 2006), which rewrites NP-SPEC into propositional formulas for testing satisfiability. The system has also been tested quite extensively in (Cadoli & Schaerf, 2005), also for several problems taken from CSPLIB, with promising results.

Interestingly, to our knowledge so far no attempt has been made to translate NP-SPEC into Answer Set Programming (ASP), which is very similar in spirit to Datalog with Circumscription, and thus a good candidate as a transformation target. Moreover, several efficient ASP software systems are available, which should guarantee good performance. A crucial advantage of ASP versus propositional satisfiability is the fact that NP-SPEC problem descriptions are in general not propositional, and therefore a reduction from NP-SPEC to SAT has to include an implicit

Preliminary versions of this work have been presented at ASPOCP 2012 and RCRA 2013.
[1]`http://www.dis.uniroma1.it/cadoli/research/projects/NP-SPEC/code/SPEC2SAT/`

instantiation (or grounding) step. Also ASP allows for variables, and ASP systems indeed provide optimized grounding procedures, which include many advanced techniques from database theory (such as indexing, join-ordering, etc). This takes the burden of instantiating in a smart way from the NP-SPEC translation when using ASP systems.

In this paper we provide a translation from NP-SPEC into ASP, using different language constructs. Our translation into ASP, together with the original SPEC2ASP rewriting, can be used to compare two different frameworks commonly used for solving problems in the complexity class NP. In particular, the aim of these translations is to implement general purpose solutions, rather than specialized algorithms and data structures. It turns out that ASP has a clear advantage in this respect thanks to its more comfortable modeling capability, even if we will also point out that more involved and specialized rewritings into SAT may result in very efficient performance. We show the correctness of the translation in a proof sketch, and discuss properties and limitations of the translation. We also provide a prototype implementation.

We then report on an extensive experimental analysis, which incorporates all previous benchmarks used for NP-SPEC, and also introduces new problem domains taken from ASP Competitions. For the latter we have created NP-SPEC encodings that are guaranteed to work with the system SPEC2SAT, which poses a number of limitations on the NP-SPEC input. For these domains we have created instances of increasing difficulty, which clearly showcase the computational advantages that a translation into ASP can provide. It turns out that this is quite independent of the choice of SAT solver, as the bottleneck occurs before the invocation of the SAT solver in the tool chain. The explanation is that translations to SAT need to include an implicit grounding, while this is delegated to ASP systems in the translation to ASP.

We will also highlight a drawback that is shared by the rewritings into SAT and ASP, which originates from the definition of the semantics of some metafacts of NP-SPEC. For example, in order to encode the guess of a partition of a relation into k sets, k different, fresh constants are introduced by the considered rewritings. It turns out that these identifiers are artificial and hence not really important for solving the input problem. Nevertheless, all current ASP instantiators as well as SPEC2SAT have to materialize k different identifiers for each tuple in the domain relation, which also means that the performance of the subsequent solving phase will be affected by the presence of obvious symmetries in the instantiated program. It is therefore our opinion that ASP may gain in terms of both practical expressivity and efficiency if constructs like partition would be introduced in the language and supported natively by solvers.

The remainder of the paper is structured as follows: in Section 2 we review the language NP-SPEC and give a very brief account of ASP. In Section 3 we provide the main ingredients for translations from NP-SPEC to ASP, and discuss properties, limitations, and correctness. In Section 4 we report on the experimental results. Finally, in Section 5 we draw our conclusions.

2. Preliminaries

This section introduces sufficient background regarding the two languages studied in this paper, namely NP-SPEC and ASP. In particular, syntax and semantics of the two languages are presented together with a few simple examples that will help to understand the similarities and differences of the two frameworks. Special attention is given here to the constructs used in NP-SPEC to specify nondeterministic guesses, which are referred to as *metafacts*.

2.1. *NP-SPEC*

We first provide a brief definition of NP-SPEC programs. For details, we refer to (Cadoli et al., 2000). We also note that a few minor details in the input language of SPEC2SAT (in which the publicly available examples are written) are different from what is described in (Cadoli et al.,

2000).

An NP-SPEC program consists of two main sections[2]: one section called DATABASE and one called SPECIFICATION, each of which is preceded by the respective keyword.

2.1.1. DATABASE

The database section defines extensional predicates or relations and (interpreted) constants. Extensional predicates are defined by writing

$$\mathrm{p} = \{\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n}\};$$

where p is a predicate symbol and each $\mathrm{t_i}$ is a tuple with matching arity. For unary predicates, each tuple is simply an integer or a constant symbol; for arity greater than 1, it is a comma-separated sequence of integers or constant symbols enclosed in parentheses. Unary extensions that are ranges of integers can also be abbreviated to $\mathrm{n..m}$, where n and m are integers or interpreted constants. Constant definitions are written as $\mathrm{c} = \mathrm{i}$; where i is an integer.

Example 1: The following defines the predicate edge representing a graph with six nodes and nine edges, and a constant n representing the number of nodes.

$$\begin{array}{l}\mathrm{DATABASE}\\ \quad \mathrm{n} = 6;\\ \quad \mathrm{edge} = \{(1,2),(3,1),(2,3),(6,2),(5,6),(4,5),(3,5),(1,4),(4,1)\};\end{array}$$

2.1.2. SPECIFICATION

The SPECIFICATION section consists of two parts: a search space declaration and a stratified Datalog program. The search space declaration serves as a domain definition for "guessed" predicates and must be one or more of the *metafacts* $\mathrm{Subset(d,p)}$, $\mathrm{Permutation(d,p)}$, $\mathrm{Partition(d,p,n)}$, and $\mathrm{IntFunc(d,p,n..m)}$, which we will describe below.

`Subset(d,p)`. This is the basic construct to which all following search space declaration constructs are reduced in the semantic definition in (Cadoli et al., 2000). Here, d is a *domain definition*, which is either an extensional predicate, a range $\mathrm{n..m}$, or a Cartesian product ($><$), union ($+$), intersection ($*$), or difference ($-$) of two domains. Symbol p is a predicate identifier and the intended meaning is that the extension of p can be any subset of the domain definition's extension, thus giving rise to nondeterminism or a "guess".

Example 2: Together with the code of Example 1, the following specification will represent all subgraphs (including the original graph) as extensions of predicate $\mathrm{subgraph}$.

$$\begin{array}{l}\mathrm{SPECIFICATION}\\ \quad \mathrm{Subset(edge, subgraph)}.\end{array}$$

`Permutation(d,p)`. Concerning this construct, d is again a domain definition, and p is a predicate identifier whose extension has the same cardinality as that of d. In particular, every tuple of the extension of p contains a distinct tuple of the extension of d, and an additional argument associating a unique integer between 1 and the cardinality of the extension of d (say,

[2]SPEC2SAT also has a third, apparently undocumented section called SEARCH, which seems to define only output features and which we will not describe here.

c), thereby defining a permutation. The extensions of p thus define bijective functions from extensions of d to $\{1..c\}$.

Example 3: Together with the code of Example 1, the following specification will represent all enumerations of edges.

SPECIFICATION
Permutation(edge, edgeorder).

One extension of edgeorder that reflects the ordering of the edges as written in Example 1 is

edgeorder(1, 2, 1), edgeorder(3, 1, 2), edgeorder(2, 3, 3),
edgeorder(6, 2, 4), edgeorder(5, 6, 5), edgeorder(4, 5, 6),
edgeorder(3, 5, 7), edgeorder(1, 4, 8), edgeorder(4, 1, 9).

`Partition(d,p,n)`. Also in this case p will have one argument more than d. In this case, extensions of p will define functions from tuples of the extension of d to $\{0..n-1\}$, thereby defining n (possibly empty) partitions.

Example 4: Together with the code of Example 1, the following specification will represent all possible pairs of graphs that partition the input graph.

SPECIFICATION
Partition(edge, partition, 2).

One extension of partition that has the first four edges in the first partition (i.e., partition 0) and the last five edges in the second partition (i.e., partition 1) would be

partition(1, 2, 0), partition(3, 1, 0), partition(2, 3, 0),
partition(6, 2, 0), partition(5, 6, 1), partition(4, 5, 1),
partition(3, 5, 1), partition(1, 4, 1), partition(4, 1, 1).

`IntFunc(d,p,n..m)`. Again, p will have one argument more than d. Here, extensions of p will define functions from tuples of the extension of d to $\{n..m\}$.

Example 5: The following specification is equivalent to the one in Example 4:

SPECIFICATION
IntFunc(edge, partition, 0..1).

Stratified Datalog Program. The stratified Datalog program is a collection of rules

$$h :- b_1, \ldots, b_m, \text{ NOT } b_{m+1}, \ldots, \text{ NOT } b_n.$$

where each $h, b_1, \ldots, b_n$ are atoms. These atoms can be of the form $p(t_1, \ldots, t_k)$ where p is a predicate symbol with arity k and all t_i are constants, variables, or arithmetic expressions formed over these. The predicates can be built-in predicates (==, <, >, >=, <=, ! =), in which case the atoms are written using infix notation. The atoms can also be aggregate atoms involving COUNT, SUM, MIN, MAX, written as for example $\text{SUM}(p(*, \_, Y), Z : n..m)$ where: $*$ specifies the argument to be aggregated over; variables that are not shared with other rule literals are local (as a special case the anonymous variable _) and represent the arguments that are not fixed; variables that are shared with other rule literals are considered fixed in the aggregation;

and variable Z will contain the valuation of the aggregate, which must be in the range $\mathrm{n..m}$. The atom h can also be the special atom fail that must not be used otherwise. This atom will always be interpreted as false, allowing for the specification of integrity constraints. The $<--$ string represents rule implication.

The rules must be stratified in the traditional sense (see for example (Apt, Blair, & Walker, 1988; Van Gelder, 1988)), meaning that there cannot be recursion through negation. Also aggregates must occur in a stratified way (see for example (Faber, Pfeifer, Leone, Dell'Armi, & Ielpa, 2008)), meaning that there cannot be recursion through aggregates. One can add comments, written in C++ style (using $/*\,*/$ or $//$).

Example 6: As an example, consider the well-known Hamiltonian Cycle problem. The NP-SPEC distribution contains an example program for an example graph:

$$
\begin{array}{l}
\mathrm{DATABASE} \\
\quad \mathrm{n} = 6;\ //\text{no. of nodes} \\
\quad \mathrm{edge} = \{(1,2),(3,1),(2,3),(6,2),(5,6),(4,5),(3,5),(1,4),(4,1)\}; \\
\mathrm{SPECIFICATION} \\
\quad \mathrm{Permutation}(\{1..\mathrm{n}\},\mathrm{path}). \\
\quad \mathrm{fail} <-- \mathrm{path}(\mathrm{X},\mathrm{P}),\ \mathrm{path}(\mathrm{Y},\mathrm{P}+1),\ \mathrm{NOT\ edge}(\mathrm{X},\mathrm{Y}). \\
\quad \mathrm{fail} <-- \mathrm{path}(\mathrm{X},\mathrm{n}),\ \mathrm{path}(\mathrm{Y},1),\ \mathrm{NOT\ edge}(\mathrm{X},\mathrm{Y}).
\end{array}
$$

The DATABASE section contains an encoding of the example graph by means of the binary predicate edge and defines a constant n for representing the number of nodes of that graph. Implicitly it is assumed that the nodes are labeled by integers from 1 to n. The SPECIFICATION section then first guesses a permutation of the nodes and then verifies the Hamiltonian Cycle condition by means of integrity constraints, one exploiting the linear order of the permutation identifiers, and another one to close the cycle from the last permutation identifier to the first one.

The semantics of NP-SPEC programs is provided in (Cadoli et al., 2000) by means of Datalog with Circumscription ($DATALOG^{CIRC}$). The syntax of this formalism consists of (positive) Datalog rules, and may also contain integrity constraints, which are written as rules containing the predicate fail in rule heads. The semantics is provided by means of $(P;Q)$-minimal models. For two Herbrand models M, N of a $DATALOG^{CIRC}$ program, $M \leq_{P;Q} N$ holds for two sets of predicates P and Q if and only if (i) predicates in Q have the same extension in M and N and (ii) for each predicate $p \in P$, the extension of p in M is a subset (possibly not proper) of the extension of p in N. A Herbrand model M of a $DATALOG^{CIRC}$ program is $(P;Q)$-minimal if there is no Herbrand model N of the program such that $N \leq_{P;Q} M$ and $M \not\leq_{P;Q} N$. This definition guarantees that only some predicates (those in P) are minimized. That means that among all models only those which are minimal with respect to predicates in P are accepted. For NP-SPEC, predicates that are defined by means of metafacts will be in the set Q. Moreover, among these only those which make the special symbol fail false are considered and referred to as answers.

An NP-SPEC program is then transformed to $DATALOG^{CIRC}$ as follows:

Each DATABASE expression $\mathrm{p} = \{\mathrm{t}_1,\ldots,\mathrm{t}_\mathrm{n}\}$ is transformed to facts $\mathrm{p}(\mathrm{t}_1)\cdots\mathrm{p}(\mathrm{t}_\mathrm{n})$, each expression $\mathrm{p} = \{\mathrm{n..m}\}$ to facts $\mathrm{p}(\mathrm{n})\cdots\mathrm{p}(\mathrm{m})$. Constant declarations such as $\mathrm{c} = \mathrm{i}$ are expanded on the fly.

Concerning SPECIFICATION expressions, for $\mathrm{Subset}(\mathrm{d},\mathrm{p})$ the domain d is materialized into a relation d; moreover, facts $\{\mathrm{out}_\mathrm{p}(\mathrm{t}_1,\ldots,\mathrm{t}_\mathrm{n}) \mid (\mathrm{t}_1,\ldots,\mathrm{t}_\mathrm{n}) \in \mathrm{H}^n \setminus d\}$, where n is the arity of d and p, and H is the Herbrand universe, and an integrity constraint $\mathrm{fail} :- \mathrm{p}(\mathrm{X}_1,\ldots,\mathrm{X}_\mathrm{n}),\mathrm{out}_\mathrm{p}(\mathrm{X}_1,\ldots,\mathrm{X}_\mathrm{n})$ are introduced. All other metafacts are reduced to the basic metafact Subset plus some rules. For the metafact $\mathrm{Permutation}(\mathrm{d},\mathrm{p})$, the metafact $\mathrm{Subset}(\mathrm{d} >< \{1..\mathrm{m}\},\mathrm{p})$ (m the cardinality of the extension of d) is translated as noted earlier,

and in addition the following rules are created:

$$\begin{array}{l}
\mathrm{p1}(\mathrm{X_1},\ldots,\mathrm{X_n}) \ :- \ \mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n},\mathrm{Z}).\\
\mathrm{fail} \ :- \ \mathrm{d}(\mathrm{X_1},\ldots,\mathrm{X_n}),\mathrm{co_{p1}}(\mathrm{X_1},\ldots,\mathrm{X_n}).\\
\mathrm{fail} \ :- \ \mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n},\mathrm{Y}),\mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n},\mathrm{Z}),\mathrm{Y}!=\mathrm{Z}.\\
\mathrm{fail} \ :- \ \mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n},\mathrm{A}),\mathrm{p}(\mathrm{Y_1},\ldots,\mathrm{Y_n},\mathrm{A}),\mathrm{X_1}!=\mathrm{Y_1}.\\
\quad\vdots\\
\mathrm{fail} \ :- \ \mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n},\mathrm{A}),\mathrm{p}(\mathrm{Y_1},\ldots,\mathrm{Y_n},\mathrm{A}),\mathrm{X_n}!=\mathrm{Y_n}.
\end{array}$$

where $\mathrm{co_{p1}}$ represents the complement of $\mathrm{p1}$ and will be defined later.

For the metafact $\mathrm{Partition(d,p,k)}$, the metafact $\mathrm{Subset(d} >< \{0..\mathrm{k}-1\},\mathrm{p})$ is translated as noted earlier, together with the following rules:

$$\begin{array}{l}
\mathrm{p1}(\mathrm{X_1},\ldots,\mathrm{X_n}) \ :- \ \mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n},\mathrm{Z}).\\
\mathrm{fail} \ :- \ \mathrm{d}(\mathrm{X_1},\ldots,\mathrm{X_n}),\mathrm{co_{p1}}(\mathrm{X_1},\ldots,\mathrm{X_n}).\\
\mathrm{fail} \ :- \ \mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n},\mathrm{Y}),\mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n},\mathrm{Z}),\mathrm{Y}!=\mathrm{Z}.
\end{array}$$

Finally, for $\mathrm{IntFunc(d,p,i..j)}$, the metafact $\mathrm{Subset(d} >< \{\mathrm{i..j}\},\mathrm{p})$ is translated as noted earlier, together with the same additional rules as for $\mathrm{Partition(d,p,k)}$.

All NP-SPEC rules are directly translated into $DATALOG^{CIRC}$ rules, replacing occurrences of $\mathrm{NOT\ p}$ by $\mathrm{co_p}$ (p being a predicate). For any predicate p (with arity n) occurring negatively in a rule (and for auxiliary $\mathrm{p1}$ predicates introduced by the translation of metafacts), the following rules are generated:

$$\begin{array}{l}
\mathrm{def_p}(\mathrm{X_1},\ldots,\mathrm{X_n}) \ :- \ \mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n}).\\
\mathrm{def_p}(\mathrm{X_1},\ldots,\mathrm{X_n}) \ :- \ \mathrm{co_p}(\mathrm{X_1},\ldots,\mathrm{X_n}).\\
\mathrm{fail} \ :- \ \mathrm{p}(\mathrm{X_1},\ldots,\mathrm{X_n}),\mathrm{co_p}(\mathrm{X_1},\ldots,\mathrm{X_n}).
\end{array}$$

What is missing is a device that ensures that def_p becomes true for all tuples of the Herbrand universe. In (Cadoli et al., 2000), this is done by means of a "restricted clause," a concise universal constraint. In order to simplify issues, we will only consider Herbrand models which contain all atoms $\mathrm{def_p}(\mathrm{t_1},\ldots,\mathrm{t_n})$ such that $\mathrm{def_p}$ has a defining rule in the translation, n is its arity, and $(\mathrm{t_1},\ldots,\mathrm{t_n}) \in \mathrm{H}^n$.

The semantics of an NP-SPEC program is then provided by the $(P;Q)$-minimal Herbrand models of the thus obtained $DATALOG^{CIRC}$ program (with the restriction of Herbrand models mentioned in the previous paragraph), where Q contains all predicates defined by metafacts and all predicates of the form co_p in the translated program, and P contains all other predicates. Let us denote the semantics of an NP-SPEC program Π by $PQMM(\Pi)$. More formally:

Definition 1*:* Let Π be an NP-SPEC program, and Π' be its translation into $DATALOG^{CIRC}$. Moreover, let $pred(\Pi')$ denote the predicates occurring in Π',

$$\begin{array}{l}
Q := \{\mathrm{p} \mid \mathrm{Subset(d,p)} \text{ occurs in } \Pi\} \cup\\
\qquad \{\mathrm{p} \mid \mathrm{Permutation(d,p)} \text{ occurs in } \Pi\} \cup\\
\qquad \{\mathrm{p} \mid \mathrm{Partition(d,p,k)} \text{ occurs in } \Pi\} \cup\\
\qquad \{\mathrm{p} \mid \mathrm{IntFunc(d,p,i..j)} \text{ occurs in } \Pi\} \cup\\
\qquad \{\mathrm{co_p} \mid \mathrm{co_p} \in pred(\Pi')\}\\
P := pred(\Pi') \setminus Q.
\end{array}$$

The semantics of Π is defined as the following set of Herbrand models:

$$PQMM(\Pi) := \{M \mid M \text{ is a } (P;Q)\text{-minimal Herbrand model of } \Pi'\}.$$

2.2. *ASP*

Concerning ASP, we only give a very brief overview, details may be found in works such as (Baral, 2003; Gebser et al., 2011; Leone et al., 2006). An ASP program consists of rules

$$\mathrm{L_1} \vee \cdots \vee \mathrm{L_k} \ :- \ \mathrm{Body}$$

where the $\mathrm{L_i}$ are literals containing variables and constants[3] (possibly containing strong negation) and Body, which is a conjunction of literals, that may also contain built-ins, aggregates and default negation. Rules without heads act like integrity constraints. The semantics is based on the Gelfond-Lifschitz reduct (Gelfond & Lifschitz, 1991) and also guarantees minimality of the answer sets. We denote the set of answer sets of an ASP program Π by $AS(\Pi)$. More formally:

Definition 2*:* Let Π be an ASP program, and I be an Herbrand interpretation. Let $ground(\Pi)$ denote the ground version of Π, obtained by replacing variables in all possible ways. The reduct of Π with respect to I, denoted Π^I, is obtained from $ground(\Pi)$ by removing all ground rules whose body is false with respect to I. I is an answer set of Π if I is a model of Π and there is no $J \subset I$ such that J is an answer set of Π. The semantics of Π is then defined as the following set of Herbrand models:

$$AS(\Pi) := \{M \mid M \text{ is an answer set of } \Pi\}.$$

Example 7: As an example, consider the Hamiltonian Cycle problem and instance from above. An ASP encoding similar to the NP-SPEC program seen earlier would be:

$$\begin{array}{l}
\#\mathrm{const\ n} = 6. \\
\mathrm{edge}(1,2).\ \mathrm{edge}(3,1).\ \mathrm{edge}(2,3).\ \mathrm{edge}(6,2).\ \mathrm{edge}(5,6). \\
\mathrm{edge}(4,5).\ \mathrm{edge}(3,5).\ \mathrm{edge}(1,4).\ \mathrm{edge}(4,1). \\
\mathrm{d}(1..\mathrm{n}). \\
\mathrm{path}(\mathrm{X},1) \vee \mathrm{path}(\mathrm{X},2) \vee \mathrm{path}(\mathrm{X},3) \vee \mathrm{path}(\mathrm{X},4) \vee \mathrm{path}(\mathrm{X},5) \vee \mathrm{path}(\mathrm{X},6) :- \mathrm{d}(\mathrm{X}). \\
:- \mathrm{path}(\mathrm{X},\mathrm{A}),\ \mathrm{path}(\mathrm{Y},\mathrm{A}),\ \mathrm{X} != \mathrm{Y}. \\
:- \mathrm{path}(\mathrm{X},\mathrm{P}),\ \mathrm{path}(\mathrm{Y},\mathrm{Z}),\ \mathrm{not\ edge}(\mathrm{X},\mathrm{Y}),\ \mathrm{Z} = \mathrm{P}+1. \\
:- \mathrm{path}(\mathrm{X},\mathrm{n}),\ \mathrm{path}(\mathrm{Y},1),\ \mathrm{not\ edge}(\mathrm{X},\mathrm{Y}).
\end{array}$$

This program is usable for gringo with clasp, using the `--shift` option (transforming the disjunctive rule into several nondisjunctive ones), and DLV. We can observe that the extensional definition is rewritten into a number of facts and that the constant definition also just changes syntax. As for the permutation statement, here we first use a predicate d representing the domain definition, and then a disjunctive rule and an integrity constraint. The disjunctive rule states that each tuple in the domain definition must be assigned one of the numbers 1 to 6, and the integrity constraint enforces the bijection, that is, no different tuples of the domain definition must be assigned the same number. The final two integrity constraints are direct translations from the NP-SPEC program. The only difference is the arithmetic expression that has been moved outside the fact in order to conform to DLV's syntax (gringo would also have accepted the immediate translation from the NP-SPEC program).

[3]Many modern ASP systems also allow for function symbols, but they are not needed here.

3. Translation from NP-SPEC to ASP

We now report how the various constructs of NP-SPEC programs can be translated into ASP. We start with the DATABASE section constructs. An extensional declaration of the form $\mathrm{p} = \{\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n}\}$ will be translated to facts $\mathrm{p}(\mathrm{t}_1) \cdots \mathrm{p}(\mathrm{t}_\mathrm{n})$, and one of the form $\mathrm{p} = \{\mathrm{n..m}\}$ will be translated to facts $\mathrm{p}(\mathrm{n}) \cdots \mathrm{p}(\mathrm{m})$. Constant declarations such as $\mathrm{c} = \mathrm{i}$, instead, will be managed in-memory by replacing all occurrences of c with i.

Now for the main task, translating the SPECIFICATION constructs. Any composed domain definition is associated with a fresh extensional predicate d as follows:

- for the Cartesian product $\mathrm{p} >< \mathrm{q}$, the following set of facts is created:

$$\{\mathrm{d}(\mathrm{x}_1, \ldots, \mathrm{x}_{\mathrm{i+j}}) \mid \mathrm{p}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i}) \wedge \mathrm{q}(\mathrm{x}_{\mathrm{i+1}}, \ldots, \mathrm{x}_{\mathrm{i+j}})\},$$

  where i and j are the arities of p and q, respectively;
- for the union $\mathrm{p} + \mathrm{q}$, the following set of facts is created:

$$\{\mathrm{d}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i}) \mid \mathrm{p}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i}) \vee \mathrm{q}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i})\},$$

  where i is the arity of both p and q;
- for the intersection $\mathrm{p} * \mathrm{q}$, the following set of facts is created:

$$\{\mathrm{d}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i}) \mid \mathrm{p}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i}) \wedge \mathrm{q}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i})\},$$

  where i is the arity of both p and q; and
- for the difference $\mathrm{p} - \mathrm{q}$, the following set of facts is created:

$$\{\mathrm{d}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i}) \mid \mathrm{p}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i}) \wedge \neg.\mathrm{q}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i})\},$$

  where i is the arity of both p and q, and $\neg.\mathrm{q}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i})$ is true if and only if the fact $\mathrm{q}(\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i})$ is not part of the translation.

For nested domain definitions, we just repeat this process recursively using fresh symbols in each recursive step. In the following we will assume that domain definitions have been treated in this way and that the top-level predicate of the translation is d and has arity n.

We then look at metafacts. The simplest one is $\mathrm{Subset}(\mathrm{d}, \mathrm{p})$, for which we produce

$$\mathrm{p}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}) \vee -\mathrm{p}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}) \ :- \ \mathrm{d}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}). \tag{1}$$

For the metafact $\mathrm{Permutation}(\mathrm{d}, \mathrm{p})$, we will create

$$\begin{array}{l} \mathrm{p}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}, 1) \vee \ldots \vee \mathrm{p}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}, \mathrm{c}) \ :- \ \mathrm{d}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}). \\ :- \ \mathrm{p}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}, \mathrm{A}), \mathrm{p}(\mathrm{Y}_1, \ldots, \mathrm{Y}_\mathrm{n}, \mathrm{A}), \mathrm{X}_1! = \mathrm{Y}_1. \\ \qquad \vdots \\ :- \ \mathrm{p}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}, \mathrm{A}), \mathrm{p}(\mathrm{Y}_1, \ldots, \mathrm{Y}_\mathrm{n}, \mathrm{A}), \mathrm{X}_\mathrm{n}! = \mathrm{Y}_\mathrm{n}. \end{array} \tag{2}$$

where n is the arity of d and c is the cardinality of d. The first rule specifies intuitively that for each tuple in d one of $\mathrm{p}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}, 1) \cdots \mathrm{p}(\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}, \mathrm{c})$ should hold, and by minimality exactly one of these will hold. The integrity constraints ensure that no different numbers will be associated to the same tuple.

The remaining metafacts are actually much simpler to translate, as the bijection criterion does not have to be checked. $\mathrm{Partition(d, p, k)}$ is translated as follows:

$$\mathrm{p(X_1, \ldots, X_n, 0) \vee \ldots \vee p(X_1, \ldots, X_n, k-1)\ :-\ d(X_1, \ldots, X_n).} \tag{3}$$

where n is the arity of d. Similarly, $\mathrm{IntFunc(d, p, i..j)}$ is translated as follows:

$$\mathrm{p(X_1, \ldots, X_n, i) \vee \ldots \vee p(X_1, \ldots, X_n, j)\ :-\ d(X_1, \ldots, X_n).} \tag{4}$$

What remains are the Datalog rules of the SPECIFICATION section. Essentially, each $\mathrm{Head} <-- \mathrm{Body}$ is directly translated into $\mathrm{Head'} :- \mathrm{Body'}$, with only minor differences. If Head is fail, then $\mathrm{Head'}$ is empty, otherwise it will be exactly the same. The difference between Body and $\mathrm{Body'}$ is due to different syntax for arithmetics, aggregates and due to safety requirements. Concerning arithmetics, gringo can accept almost the same syntax as NP-SPEC with only minor differences ($\#\mathrm{abs}$ instead of abs, $\#\mathrm{pow}$ instead of $\hat{}$), while DLV is much more restrictive. DLV currently does not support negative integers and it does not provide constructs corresponding to $\hat{}$. Moreover, arithmetic expressions may not be nested in DLV programs, but this limitation can be overcome by flattening the expressions.

Concerning aggregates, DLV and gringo support similar syntax, which is a little bit different from the one used in NP-SPEC but rather straightforward to rewrite according to the following schema: Arguments marked with asterisks are first replaced with fresh variables; these are the arguments on which the aggregation function is applied. Apart from COUNT, exactly one asterisk may appear in each aggregate. Hence, an aggregate $\mathrm{SUM(p(*, \_, Y), Z : n..m)}$ is written as

$$\mathrm{\#sum\{X : p(X, \_, Y)\} = Z,\ d(Z)}$$

where X is a fresh variable and d is a fresh predicate defined by facts $\mathrm{d(n)} \cdots \mathrm{d(m)}$. Aggregates MIN and MAX are rewritten similarly, while $\mathrm{COUNT(p(*, \_, *, Y), Z : n..m)}$ is written as

$$\mathrm{\#count\{X_1, X_2 : p(X_1, \_, X_2, Y)\} = Z,\ d(Z)}$$

A more difficult problem presents the safety conditions enforced by the ASP systems. NP-SPEC has a fairly lax safety criterion, while for instance DLV requires each variable to occur in a positive, non-builtin body literal, and also gringo has a similar criterion. This mismatch can be overcome by introducing appropriate domain predicates when needed, we omit the details for clarity and will assume in the following that only safe NP-SPEC rules are used.

Given an NP-SPEC program Π, we denote by Π^* the translation to ASP.

Theorem 3.1: *For an NP-SPEC program* Π*, there is a one-to-one correspondence between* $PQMM(\Pi)$ *and* $AS(\Pi^*)$.

3.1. *Proof of Theorem 3.1*

Throughout this section, let Π be an NPSPEC program, and Π' be the $DATALOG^{CIRC}$ program associated with Π. We first prove that each $(P; Q)$-minimal model of Π' can be mapped into an answer set of Π^*. In particular, we provide the mapping function in the next definition. We then we show that the output of the mapping is a model of Π^*, and that it is also minimal for the associated reduct.

Definition 3*:* Let $M \in PQMM(\Pi)$. We define

$$\begin{aligned} M^* = (M \setminus & \{\mathrm{out_p}(\bar{\mathrm{t}}) \mid \mathrm{out_p}(\bar{\mathrm{t}}) \in M\} \\ & \setminus \{\mathrm{co_p}(\bar{\mathrm{t}}) \mid \mathrm{co_p}(\bar{\mathrm{t}}) \in M\} \\ & \setminus \{\mathrm{def_p}(\bar{\mathrm{t}}) \mid \mathrm{def_p}(\bar{\mathrm{t}}) \in M\} \\ & \setminus \{\mathrm{p1}(\bar{\mathrm{t}}) \mid \mathrm{p}(\bar{\mathrm{t}}, \mathrm{s}) \in M\}) \\ & \cup \{-\mathrm{p}(\bar{\mathrm{t}}) \mid -\mathrm{p} \text{ is a predicate in } \Pi^*,\ \mathrm{p}(\bar{\mathrm{t}}) \notin M\}. \end{aligned}$$

Lemma 3.2: *Let $M \in PQMM(\Pi)$. M^* is a model of Π^*.*

Proof. For translations of NP-SPEC rules, this holds because any positive body literals and head atoms in Π' are true in M if and only if they are true in M^*. Any negative literal $\mathrm{NOT\ a}(\bar{\mathrm{t}})$ in Π' is true in M if and only if $\mathrm{co_a}(\bar{\mathrm{t}}) \in M$ and $\mathrm{a}(\bar{\mathrm{t}}) \notin M$, therefore $\mathrm{a}(\bar{\mathrm{t}}) \notin M^*$ and $\mathrm{not\ a}(\bar{\mathrm{t}})$ is true in M^*. For constraints, this holds because $\mathrm{fail} \notin M$.

For translations of NP-SPEC metafacts, we provide detailed comments below.

- For $\mathrm{Subset(d, p)}$, the associated rule (1) is satisfied because M obeys the excluding middle law. In particular, since M satisfies $\mathrm{Subset(d, p)}$, if $\mathrm{p}(\bar{\mathrm{t}}) \in M$ then $\mathrm{d}(\bar{\mathrm{t}}) \in M$, which in turn implies $\mathrm{p}(\bar{\mathrm{t}}) \in M^*$. On the other hand, for each $\mathrm{d}(\bar{\mathrm{t}}) \in M$ such that $\mathrm{p}(\bar{\mathrm{t}}) \notin M$, M^* contains $-\mathrm{p}(\bar{\mathrm{t}})$.
- For $\mathrm{Permutation(d, p)}$ we observe that for each tuple $\bar{\mathrm{t}}$ in d there is exactly one $\mathrm{p}(\bar{\mathrm{t}}, \mathrm{s})$ in M and M^* such that $s \in [1..m]$ where m is the cardinality of the extension of d. Hence the disjunctive rule in (2) is satisfied. Moreover, the constraints in (2) are equal to the $DATALOG^{CIRC}$ constraints in Π' and thus also satisfied.
- For $\mathrm{Partition(d, p, k)}$ we observe that for each tuple $\bar{\mathrm{t}}$ in d there is exactly one $\mathrm{p}(\bar{\mathrm{t}}, \mathrm{s})$ in M and M^* such that $s \in [0..k-1]$, hence rule (3) in Π^* is satisfied.
- Similarly, for $\mathrm{IntFunc(d, p, k)}$ for each tuple $\bar{\mathrm{t}}$ in d there is exactly one $\mathrm{p}(\bar{\mathrm{t}}, \mathrm{s})$ in M and M^* such that $s \in [i..j]$, hence rule (4) in Π^* is satisfied.

□

Theorem 3.3: *Let $M \in PQMM(\Pi)$. M^* is an answer set of Π^*.*

Proof. By the previous lemma, it remains to show that M^* is also a minimal model of the reduct $(\Pi^*)^{M^*}$. Assume by contradiction that there is $N \subset M^*$ such that N is a model of $(\Pi^*)^{M^*}$, and consider interpretation

$$\begin{aligned} N' := M \setminus & (M^* \setminus N) \\ & \setminus \{\mathrm{p1}(\bar{\mathrm{t}}) \mid \mathrm{p1} \text{ is a predicate in } \Pi',\ \mathrm{p}(\bar{\mathrm{t}}) \in (M^* \setminus N)\}. \end{aligned}$$

Our aim is to show that N' is a model of Π, which would contradict $M \in PQMM(\Pi)$ because $N' \leq_{P;Q} M$ and $M \not\leq_{P;Q} N'$. We start by observing that rules in $(\Pi^*)^{M^*}$ modeling metafacts are such that all predicates associated with metafacts have the same extension in N and M^*. Indeed, this is true by construction of M^* because M is a $(P;Q)$-minimal Herbrand model of Π' by assumption, and therefore each ground instance of a disjunctive rules in (1)–(4) is satisfied either because of a false body literal, or because of a *unique* true head atom. The remaining predicates are defined by Datalog rules, which have a one-to-one corresponding into Π'. We can thus conclude that N' is a model of Π, i.e., a contradiction. □

We now provide the mapping on the other direction, i.e., a function associating any answer set of Π^* with a $(P;Q)$-minimal model of Π'.

Definition 4*:* Let $A \in AS(\Pi^*)$. We define

$$\begin{aligned} A^\circ = (A \setminus \{-\mathrm{p}(\bar{\mathrm{t}}) \mid -\mathrm{p} \text{ is a predicate in } \Pi^*\}) \\ \cup \{\mathrm{out_p}(\bar{\mathrm{t}}) \mid \mathrm{Subset(d,p)} \in \Pi', \bar{\mathrm{t}} \in \mathrm{H}^{|\mathrm{p}|}, \mathrm{d}(\bar{\mathrm{t}}) \notin A\} \\ \cup \{\mathrm{out_p}(\bar{\mathrm{t}}, \mathrm{s}) \mid \mathrm{Permutation(d,p)} \in \Pi', \bar{\mathrm{t}} \in \mathrm{H}^{|\mathrm{d}|}, s \in [1..c(\mathrm{d})], \mathrm{d}(\bar{\mathrm{t}}) \notin A\} \\ \cup \{\mathrm{out_p}(\bar{\mathrm{t}}, \mathrm{s}) \mid \mathrm{Partition(d,p,k)} \in \Pi', \bar{\mathrm{t}} \in \mathrm{H}^{|\mathrm{d}|}, s \in [1..k], \mathrm{d}(\bar{\mathrm{t}}) \notin A\} \\ \cup \{\mathrm{out_p}(\bar{\mathrm{t}}, \mathrm{s}) \mid \mathrm{IntFunc(d,p,i..j)} \in \Pi', \bar{\mathrm{t}} \in \mathrm{H}^{|\mathrm{d}|}, s \in [i..j], \mathrm{d}(\bar{\mathrm{t}}) \notin A\} \\ \cup \{\mathrm{p1}(\bar{\mathrm{t}}) \mid \mathrm{p}(\bar{\mathrm{t}}, \mathrm{s}) \in A\} \\ \cup \{\mathrm{co_p}(\bar{\mathrm{t}}) \mid \mathrm{p} \text{ occurs negated in } \Pi', \bar{\mathrm{t}} \in \mathrm{H}^{|\mathrm{p}|}, \mathrm{p}(\bar{\mathrm{t}}) \notin A\} \\ \cup \{\mathrm{def_p}(\bar{\mathrm{t}}) \mid \mathrm{p} \text{ occurs negated in } \Pi', \bar{\mathrm{t}} \in \mathrm{H}^{|\mathrm{p}|}\} \end{aligned}$$

where $c(\mathrm{d})$ is the cardinality of the extension of d.

Theorem 3.4: *Let $A \in AS(\Pi^*)$. Then $A^\circ \in PQMM(\Pi)$.*

Proof. A° is an Herbrand model of Π' by construction. Assume by contradiction that there is another Herbrand model B of Π' such that $B \leq_{P;Q} A^\circ$ and $A^\circ \not\leq_{P;Q} B$. It can be shown that

$$B' := A \setminus (A^\circ \setminus B)$$

is a model of $(\Pi^*)^A$ such that $B' \subset A$, which is a contradiction. Indeed, predicates defined by metafacts have the same extension in A° and in B by assumption. The same observation also applies to predicates of the form $\mathrm{co_p}(\bar{\mathrm{t}})$, which means that the interpretation of negative literals is fixed in $(\Pi^*)^A$. The remaining predicates are defined by Datalog rules in Π', and have one-to-one counterparts in Π^*. Each rule of this kind whose body is true with respect to both A° and B is such that the head is true with respect to both A° and B as well. Therefore, the corresponding rule in Π^* is such that the head is true with respect to B', which completes our proof. □

3.2. *Alternative translations*

In this section we provide a brief description of an alternative translation using aggregates and choice rules. We start with the metafact $\mathrm{Subset(d,p)}$, which can be translated as follows:

$$\{\mathrm{p}(\mathrm{X_1}, \ldots, \mathrm{X_n}) : \mathrm{d}(\mathrm{X_1}, \ldots, \mathrm{X_n})\}. \tag{5}$$

Concerning the metafact $\mathrm{Permutation(d,p)}$, the disjunctive rule can be replaced by the following choice rule:

$$1\{\mathrm{p}(\mathrm{X_1}, \ldots, \mathrm{X_n}, 1..\mathrm{c})\}1 :- \mathrm{d}(\mathrm{X_1}, \ldots, \mathrm{X_n}). \tag{6}$$

while the n integrity constraints can be replaced by just one using an aggregate:

$$:- \#\mathrm{count}\{\mathrm{X_1}, \ldots, \mathrm{X_n} : \mathrm{p}(\mathrm{X_1}, \ldots, \mathrm{X_n}, \mathrm{A})\} > 1,\ \mathrm{p}(\_, \ldots, \_, \mathrm{A}). \tag{7}$$

For the metafact $\mathrm{Partition(d,p,k)}$, the following choice rule can be used

$$1\{\mathrm{p}(\mathrm{X_1}, \ldots, \mathrm{X_n}, 0..\mathrm{k}-1)\}1 :- \mathrm{d}(\mathrm{X_1}, \ldots, \mathrm{X_n}). \tag{8}$$

while the metafact $\mathrm{IntFunc(d,p,i..j)}$ can be transformed into

$$1\{\mathrm{p}(\mathrm{X_1}, \ldots, \mathrm{X_n}, \mathrm{i..j})\}1 :- \mathrm{d}(\mathrm{X_1}, \ldots, \mathrm{X_n}). \tag{9}$$

```
DATABASE
    n = 10;
    manAssignsScore = {(1,1,2), (1,2,1), ...};
    womanAssignsScore = {(1,1,2), (1,2,2), ...};

SPECIFICATION
    IntFunc({1..n}, match, 1..n).

    fail <-- match(M1,W), match(M,W), M <> M1.

    fail <-- match(M,W1), manAssignsScore(M,W,Smw), W1 <> W,
             manAssignsScore(M,W1,Smw1),   Smw >  Smw1,
             match(M1,W), womanAssignsScore(W,M,Swm),
             womanAssignsScore(W,M1,Swm1), Swm >= Swm1.
```

Figure 1. Encoding of stable marriage

In order to show the equivalence of the alternative translations with the first translation introduced in this section, we observe that the rewritten programs are *head-cycle-free* (Ben-Eliyahu & Dechter, 1994). For such programs disjunction can be eliminated by means of a procedure known as *shift* (Eiter, Fink, & Woltran, 2007), which essentially replaces a disjunctive rule of the form:

$$\alpha_1 \vee \cdots \vee \alpha_n :- body.$$

into n rules of the form:

$$\alpha_i :- body,\ not\ \alpha_1,\ \ldots,\ not\ \alpha_{i-1},\ not\ \alpha_{i+1},\ \ldots,\ not\ \alpha_n.$$

one for each $i \in [1..n]$. According to the ASP Core 2 standard (Calimeri, Ianni, & Ricca, 2014), the rules above defines the semantics of a choice rule of the form:

$$\{\alpha_1;\ \cdots\ ;\ \alpha_n\} :- body.$$

from which we derive the equivalence of (1) and (5). Similarly, the semantics of a choice rule of the form:

$$l\{\alpha_1;\ \cdots\ ;\ \alpha_n\}u :- body.$$

is defined by the following rules:

$$\begin{array}{l}\{\alpha_1;\ \cdots\ ;\ \alpha_n\} :- body.\\ :- l \leq \#count\{\alpha_1;\ \cdots\ ;\ \alpha_n\} \leq u.\end{array}$$

from which we derive the equivalence of (2) and (6)–(7), (3) and (8), and (4) and (9).

4. Experiments

We have created a prototype implementation of the transformation described in Section 3, which is available at `http://archives.alviano.net/npspec2asp/`. It is written in `C++` using `bison` and `flex`, and called NPSPEC2ASP. The implementation does only rudimentary

```
DATABASE
    edge = {(0,6), (0,3), (0,4), (1,5), ...};
    node = {1, 0, 3, 2, 5, 4, 7, 6, 9, 8};
    num_edges = 42;

SPECIFICATION
    IntFunc(node, value, 0..num_edges).

    edge_value(X,Y,V1-V2) <-- edge(X,Y), value(X,V1), value(Y,V2), V1 >= V2.
    edge_value(X,Y,V2-V1) <-- edge(X,Y), value(X,V1), value(Y,V2), V2 > V1.

    fail <-- value(X,N), value(Y,N), X<Y.

    fail <-- edge_value(X,Y,N1), edge_value(X,Y,N2), N1 != N2.

    fail <-- edge_value(X1,Y1,N), edge_value(X2,Y2,N), X1 != X2.
    fail <-- edge_value(X1,Y1,N), edge_value(X2,Y2,N), Y1 != Y2.
```

Figure 2. Encoding of graceful graphs

correctness checks of the program and is focused on generating ASP programs for correct NP-SPEC input. It generates either the disjunctive rules or the choice rules described in Section 3. For the experiments, the transformation used for $\mathrm{Permutation}$ produced the integrity constraint with the counting aggregate. We used this implementation to test the viability of our approach, in particular assessing the efficiency of the proposed rewriting into ASP with respect to the previously available transformation into SAT.

4.1. *Benchmark settings*

In the benchmark we included several instances available on the NP-SPEC site. More specifically, we considered two sets of instances, namely the *miscellanea* and *csplib2npspec* benchmarks. Even if these instances have been conceived for demonstrating the expressivity of the language rather than for assessing the efficiency of an evaluator, it turned out that even for these comparatively small instances there are quite marked performance differences.

We also considered benchmarks from the 3rd and 4th ASP Competitions (Calimeri et al., 2011; Alviano et al., 2013). In particular, we tested *bottle filling*, *graceful graphs*, *Hamiltonian cycle*, and *stable marriage*. For these domains we generated instances smaller in size than those used for the competitions. This is motivated by a compromise we had to do in order to test SPEC2SAT. In fact, it seems that SPEC2SAT does not support recursive rules, which often requires to write encodings that are intrinsically inefficient. Just to make an example, the encoding of Hamiltonian cycle available on the web site of SPEC2SAT and reported in Example 6 guesses a permutation of the nodes on the input graphs. The search space has thus size $n!$, where n is the number of nodes in the input graph. A more efficient encoding instead would guess a subset of the edges, thus defining a significantly smaller search space of size 2^m, where m is the number of edges. The other tested encodings are reported in Figures 1–3. We observe that the encodings of stable marriage and graceful graphs use the metafact $\mathrm{IntFunc}$, while the encoding of bottle filling uses Subset and aggregates.

The experiment was executed on an Intel Xeon CPU X3430 2.40GHz with 4 GB of central memory, running Debian 7.2 with kernel Linux 3.2.0-4-amd64. Memory was limited to 3 GB and time to 600 seconds. The tool NPSPEC2ASP was compiled with gcc 4.8.2. The other tools involved in the experiment are SPEC2SAT 1.1 (Cadoli & Schaerf, 2005), satz 215.2 (Li, 1999),

```
DATABASE
    rows = 6;
    cols = 6;
    xsucc = {(0,1), (1,2), (2,3), (3,4), (4,5)};
    ysucc = {(0,1), (1,2), (2,3), (3,4), (4,5)};
    xvalue = {(1,2), (2,1), (3,3), (4,2), (5,0)};
    yvalue = {(1,1), (2,0), (3,3), (4,2), (5,2)};
    bottle = {(1,1,4), (2,3,2), (2,2,2), (2,3,3), ...};
    bottle_position = {(1,4), (3,2), (2,2), (3,3), ...};

SPECIFICATION
    Subset(bottle_position, filled).

    fail <-- xvalue(Y,V), COUNT(filled(*,Y),C:0..cols), C <> V.
    fail <-- yvalue(X,V), COUNT(filled(X,*),C:0..rows), C <> V.

    fail <-- bottle(B,X1,Y1), bottle(B,X2,Y2), ysucc(Y1,Y2),
             filled(X1,Y1), NOT filled(X2,Y2).
    fail <-- bottle(B,X1,Y), bottle(B,X2,Y), filled(X1,Y), NOT filled(X2,Y), X1 <> X2.
```

Figure 3. Encoding of bottle filling

minisat 2.2 (Eén & Sörensson, 2003), gringo 3.0.5[4] (Gebser, Schaub, & Thiele, 2007), clasp 2.1.5 (Gebser, Kaufmann, Neumann, & Schaub, 2007), cmodels 3.85 (Lierler & Maratea, 2004), DLV 2012-12-17 (Alviano et al., 2011), and wasp 1.0 (Dodaro et al., 2011).

In our experiments, we first measured the running time required by SPEC2SAT and NPSPEC2ASP to rewrite the input specification into SAT and ASP, respectively. Then, for each SAT encoding produced by SPEC2SAT, we ran three SAT solvers, namely satz, minisat and clasp, to obtain one solution if one exists. For each of these executions we measured the time to obtain the solution or the assertion that none exists, thus the sum of the running times of SPEC2SAT and of the SAT solvers. Moreover, for each ASP encoding produced by NPSPEC2ASP, we ran two instantiators, namely gringo and DLV (with option `-instantiate`). Actually, for DLV we also tested a slightly different version producing ground programs in numeric format, i.e., DLV^w (`https://www.mat.unical.it/ricca/wasp/`), and for gringo and the associated solvers we also tested the rewriting into choice rules. For each of these runs we measured the time required to compute the ground ASP program, thus the sum of the running times of NPSPEC2ASP and of the instantiator. Finally, for each ground ASP program, we computed one solution by using clasp, cmodels, DLV and wasp, and measured the overall time required by the tool-chain. For the miscellanea and csplib2npspec benchmarks we have also measured the sizes of the instantiated formulas and programs. For SPEC2SAT, we report the number of clauses in the produced formula and the number of propositional variables occurring in it. For DLV and gringo we report the number of ground rules produced and the number of ground atoms occurring in them. There is a slight difference in the statistics provided by DLV and gringo: DLV does not count ground atoms (and facts) that were already found to be true; to be more comparable, we added the number of facts for DLV.

4.2. *Benchmarks from the NP-SPEC site*

Experimental results concerning the *miscellanea* and *csplib2npspec* benchmarks are reported in Table 1, where the time required by NPSPEC2ASP has been omitted because it is always below the measurement accuracy. On the other hand, the execution time of SPEC2SAT is higher,

[4]We did not use gringo 4, as it is still preliminary and at the time of testing did not provide some of the functionality of gringo 3, in our case the `--shift` option that we used in the benchmarks.

sometimes by several orders of magnitude, with a peak on *golombRuler* for which SPEC2SAT did not terminate on the allotted time. In fact, SPEC2SAT has to compute a ground SAT instance to pass to a SAT solver, while NPSPEC2ASP outputs a non-ground ASP program. In fact, it is more meaningful to compare SPEC2SAT to NPSPEC2ASP plus the ASP instantiator to obtain a ground ASP program. Columns gringo and "DLV inst" report these times, which are however always less than those of SPEC2SAT. However, we would like to point out that this is just a rough comparison, as SPEC2SAT obviously performs a different computation (with different output) than NPSPEC2SAT plus an ASP instantiator. For a similar reason we do not provide a comparison between SAT and propositional ASP solvers: these systems even start from different input.

In Table 2 it can be seen that also the number of ground rules produced by the ASP systems is usually smaller than the number of clauses produced by SPEC2SAT, even if often the number of ground atoms exceeds the number of propositional variables. These numbers are to be compared with the same caveat as the timings: one should be aware of the fact that one figure refers to propositional formulas, the other to logic programming rules, so they are not directly comparable.

Concerning the computation of one solution from each ground specification, all considered SAT and ASP solvers are fast in almost all tests. Among the exceptions are satz for *proteinFolding*, which exceeds the allotted time, and DLV for *jobShopScheduling*, whose execution lasted around 94 seconds. A hard instance is *allInterval*, for which only satz, DLV and DLV^w+wasp terminated in the allotted time. All other solvers, including gringo+clasp and gringo+cmodels, exceeded the allotted time, even if the NPSPEC2ASP rewriting and the instantiation by gringo is produced in less time than the output of SPEC2SAT. This instance is an outlier in our experiments and we conjecture that it is due to an "unlucky case" for the heuristics adopted by minisat, clasp and cmodels. In almost all other instances the NPSPEC2ASP toolchains compute solutions in less than 1 second, while SPEC2SAT toolchains typically require several seconds, see in particular *langford*, *lowAutocorrelation* and *magicSquare*. For this last instance we also measured a timeout for gringo+cmodels. The size of the programs produced by the ASP instantiators is always smaller than the size of the formulas produced by SPEC2SAT, sometimes by orders of magnitude, even if the number of ground atoms often exceeds the number of propositional variables. A major cause for the difference in size appear to be aggregates in the problem specification, which are supported natively by ASP systems, but require expensive rewritings for SPEC2SAT.

4.3. *Benchmarks from the 3rd and 4th ASP Competitions*

Figures 4–8 reports the average execution times measured for the other benchmarks in our experiment. For three of these benchmarks we also tested handwritten translations for co-

Table 1. Running times on the *miscellanea* and *csplib2npspec* benchmarks

Instance	SPEC2SAT				NPSPEC2ASP						
	SPEC2SAT alone	+satz	+minisat	+clasp	DLV inst	DLV	DLV^w inst	DLV^w +wasp	gringo	gringo +clasp	gringo+ cmodels
allInterval	1.14	44.16	267.14	>600	0.00	0.90	0.00	0.10	0.00	>600	>600
bacp	5.25	5.20	5.19	5.21	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bibd	3.33	3.41	3.34	3.35	0.00	0.00	0.00	0.00	0.00	0.00	0.21
carSequencing	7.61	15.02	7.65	7.61	1.36	1.36	0.90	1.02	0.10	0.10	1.09
factoring	5.15	8.00	5.20	5.95	0.22	0.45	0.33	0.43	0.00	1.01	14.19
golombRuler	>600	>600	>600	>600	58.29	58.84	37.45	41.44	7.49	9.51	16.21
jobShopScheduling	36.57	37.82	36.89	37.33	3.19	90.73	1.93	3.84	0.22	1.12	4.72
langford	9.65	10.65	9.80	10.19	0.00	0.79	0.00	0.61	0.00	0.00	16.13
lowAutocorrelation	19.24	20.04	19.29	19.57	N/A*	N/A*	N/A*	N/A*	0.00	0.00	0.00
magicSquare	10.34	10.57	10.39	10.34	0.10	18.87	0.10	1.18	0.00	0.04	>600
proteinFolding	113.44	>600	114.11	115.08	N/A*	N/A*	N/A*	N/A*	0.91	2.19	7.20
socialGolfer	6.20	6.37	6.23	6.17	0.00	0.10	0.10	0.10	0.00	0.00	0.08
sudoku	2.52	2.53	2.53	2.53	0.10	0.10	0.10	0.20	0.00	0.00	0.05

* The instance contains negative integers.

pris (`http://bach.istc.kobe-u.ac.jp/copris/`), a state-of-the-art CSP-to-SAT grounder (Tamura, Taga, Kitagawa, & Banbara, 2009), using both minisat and clasp as SAT solver backends. Copris was included in order to provide an idea how modern CSP-to-SAT techniques fare with respect to SPEC2SAT. However, there are some crucial differences between the language accepted by copris and NP-SPEC, most importantly the apparent absence of language features supporting inductive definitions in copris (provided by Datalog rules in NP-SPEC; techniques introduced in (Pelov & Ternovska, 2005; Mariën, Wittocx, Denecker, & Bruynooghe, 2008) may be used for this purpose, but are not the objective of this paper). On the other hand, SPEC2SAT also does not support recursive predicate definitions in the NP-SPEC input (but NPSPEC2ASP does). Plots of copris do not include Scala-to-Java compilation time.

In the graphs, instances are sorted by size and systems have been run up to the first timeout. In the figures, graphs on top report grounding times, while grounding+solving times are shown below. A few comments on these graphs are reported below.

Figure 4 reports the result for stable marriage and Figure 5 those for graceful graphs, both of whose encodings do not include aggregates. Concerning the grounding, gringo and DLV are significantly more efficient than SPEC2SAT, with gringo performing better than DLV in graceful graphs. As for the solving, we first observe that the tested SAT solvers solved the Boolean formulas produced by SPEC2SAT in less than 1 second. Hence, our comparison of the solving times is mainly focused on the different ASP solvers. In this respect, for stable marriage we note that both clasp and cmodels are in general faster on the encoding using choice rules. For graceful graphs, DLV solved only the first 5 instance sizes, while the other systems performed significantly better. For this benchmark we also observe that the performance of cmodels does not improve with the encoding using choice rules, while clasp has a different behavior. Concerning copris, we note that it outperforms other systems on graceful graphs, while the opposite happens for stable marriage.

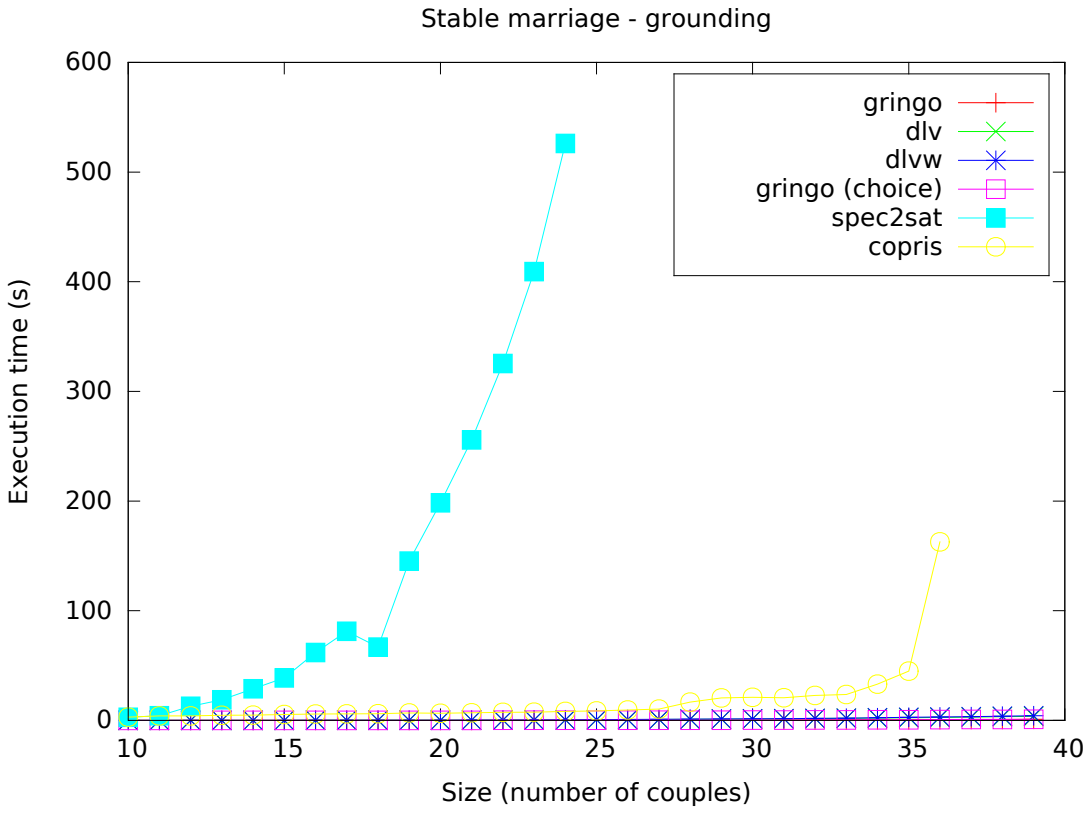
For bottle filling we tested an encoding including aggregates. The result for this benchmark is reported in Figures 6 and 7. Satisfiable and unsatisfiable instances are shown in different graphs in this case. Concerning the grounding, we observe again a clear advantage of gringo and DLV over SPEC2SAT. In fact, SPEC2SAT runs into time out for satisfiable instances of size 25 and unsatisfiable instances of size 24. Moreover, we have also to point out that the output of SPEC2SAT is always an unsatisfiable Boolean formula for these testcases, which appears to be due to a bug in the instantiation of aggregates. Hence, for the solving phase we only tested ASP solvers. DLV can solve only the smallest instances, up to grids of size 10-11. Wasp instead can solve instances up to grids of size 19, and cmodels solved instances of size 22. The performance of cmodels improves on the rewriting based on choice rules. We finally point out that clasp outperforms all other systems in this benchmark, solving all tested instances in a few tenths of a

Table 2. Instance sizes of the *miscellanea* and *csplib2npspec* benchmarks

Instance	SPEC2SAT		NPSPEC2ASP					
			DLV		DLV^w		gringo	
	Clauses	Variables	Rules	Atoms	Rules	Atoms	Rules	Atoms
allInterval	21,737	761	9,239	1,639	9239	1639	9,961	1,601
bacp	39,531	1,518	314	316	322	392	436	360
bibd	31,843	4,424	2,684	2,047	2705	2404	4,091	2,279
carSequencing	39,875	786	33,398	219	33428	303	33,506	218
golombRuler	N/A**	N/A**	653,593	96	653,610	96	1,149,561	105
jobShopScheduling	209,495	1,980	156,107	2,052	156,287	2,232	158,087	2,089
langford	130,518	7299	3,736	793	3574	1054	4,015	803
lowAutocorrelation	186,407	5,952	N/A*	N/A*	N/A*	N/A*	2,339	1,041
magicSquare	38,564	1,975	5458	872	5773	1085	18,445	14,513
proteinFolding	735,721	669	N/A*	N/A*	N/A*	N/A*	520,107	347
socialGolfer	21,600	1,424	11,097	441	11105	561	11,321	442
sudoku	33,825	1,458	24,777	2,545	25,962	2545	25,263	1,736

* The instance contains negative integers.

** The system did not terminate in 30 minutes.



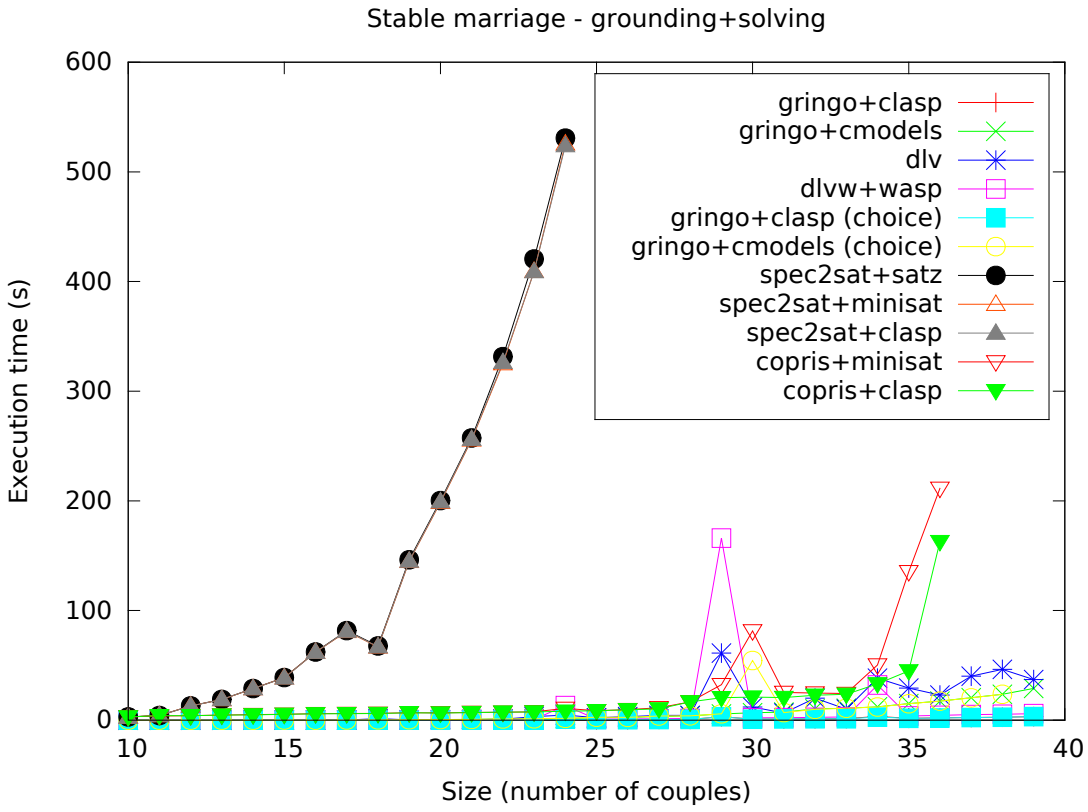


Figure 4. Average running time on stable marriage

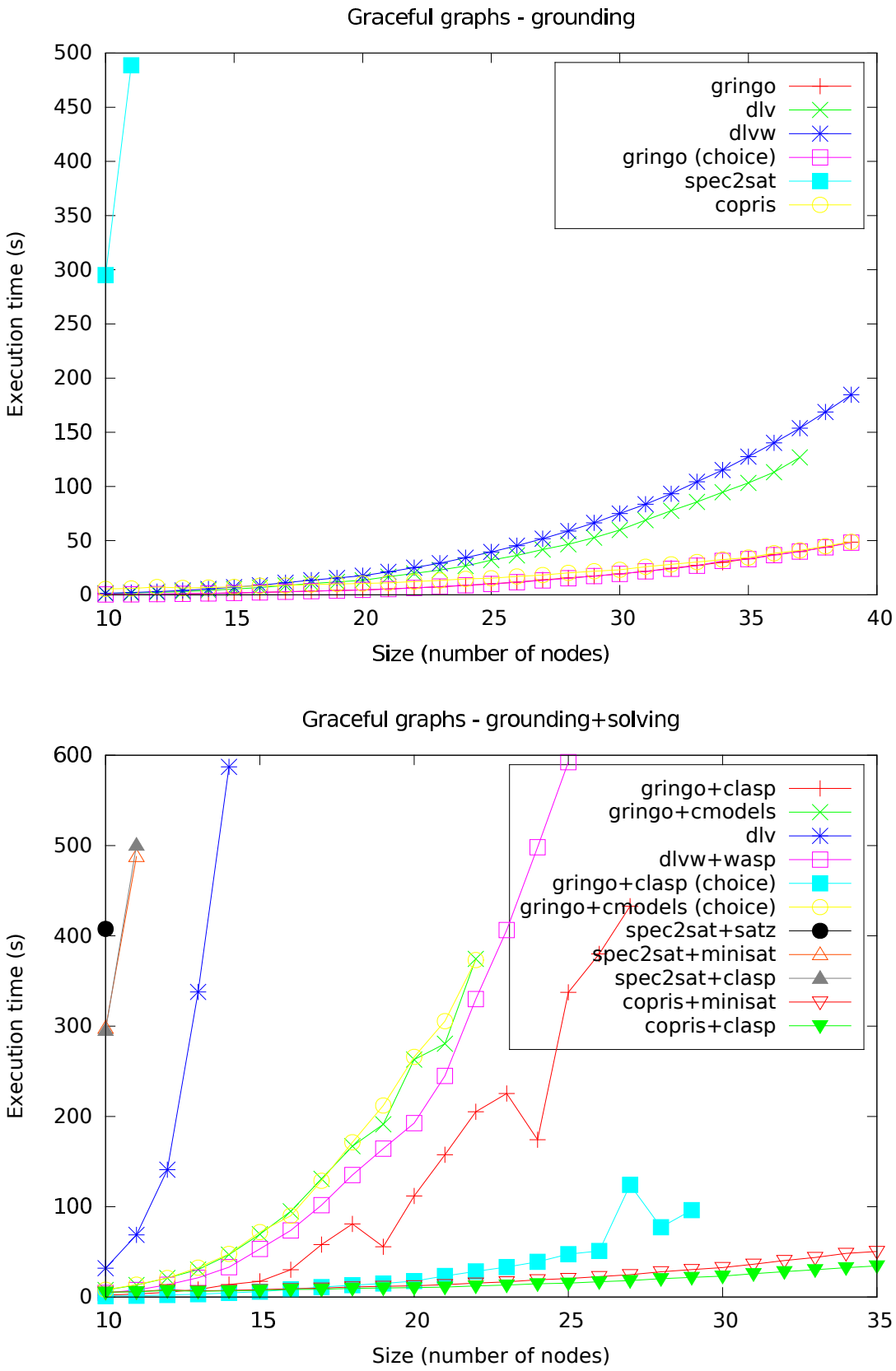


Figure 5. Average running time on graceful graphs

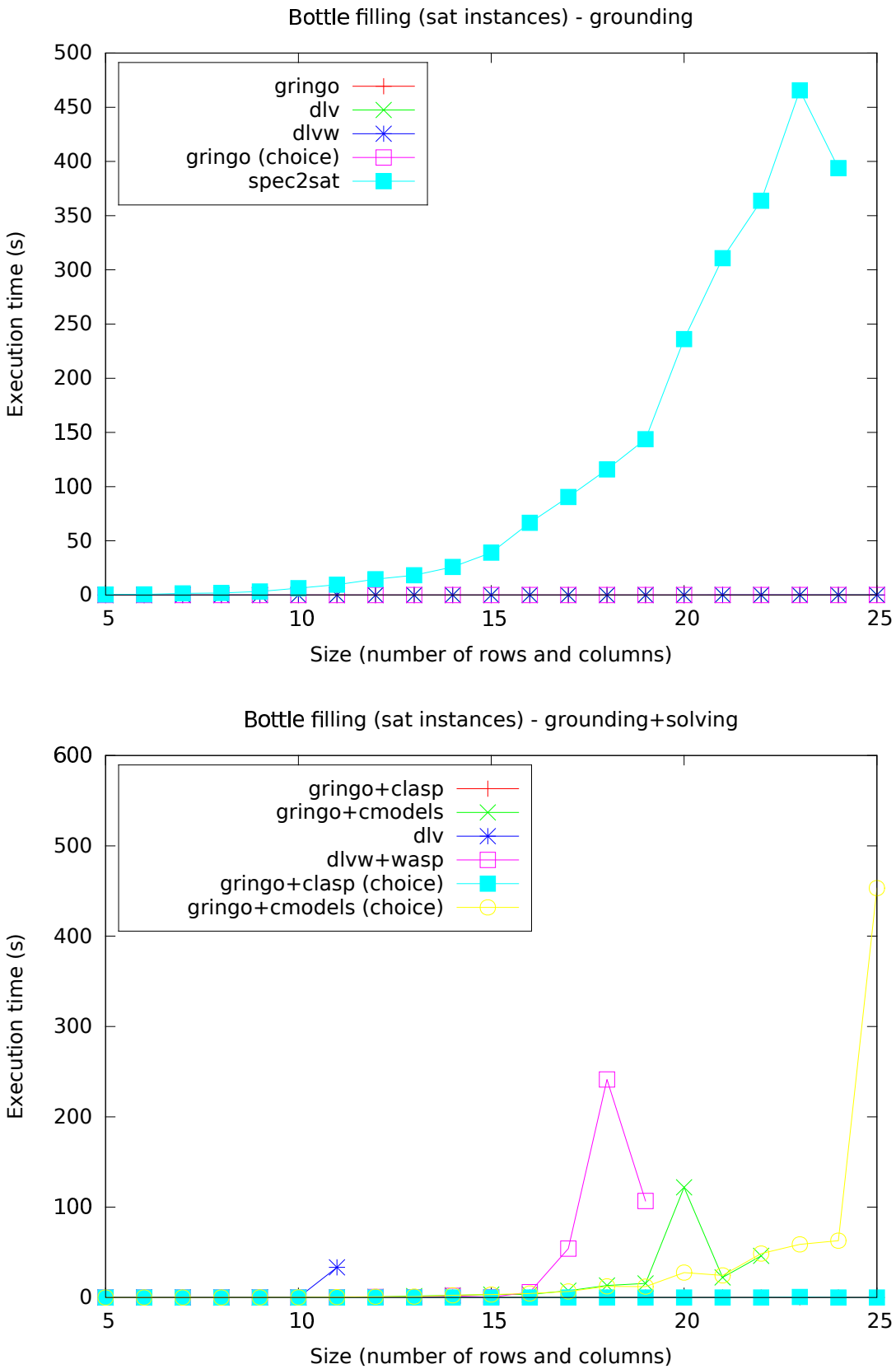


Figure 6. Average running time on satisfiable bottle filling

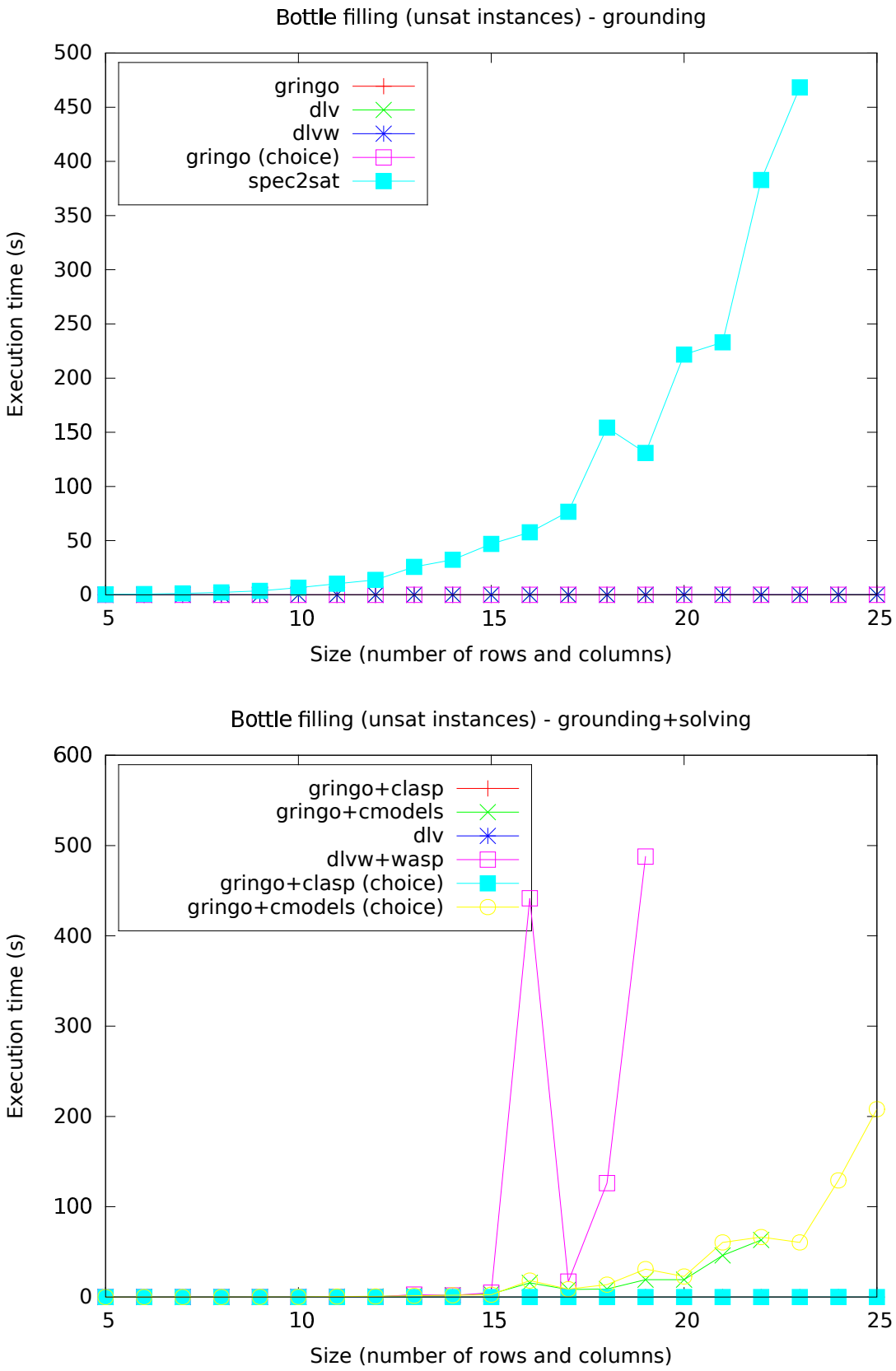


Figure 7. Average running time on unsatisfiable bottle filling

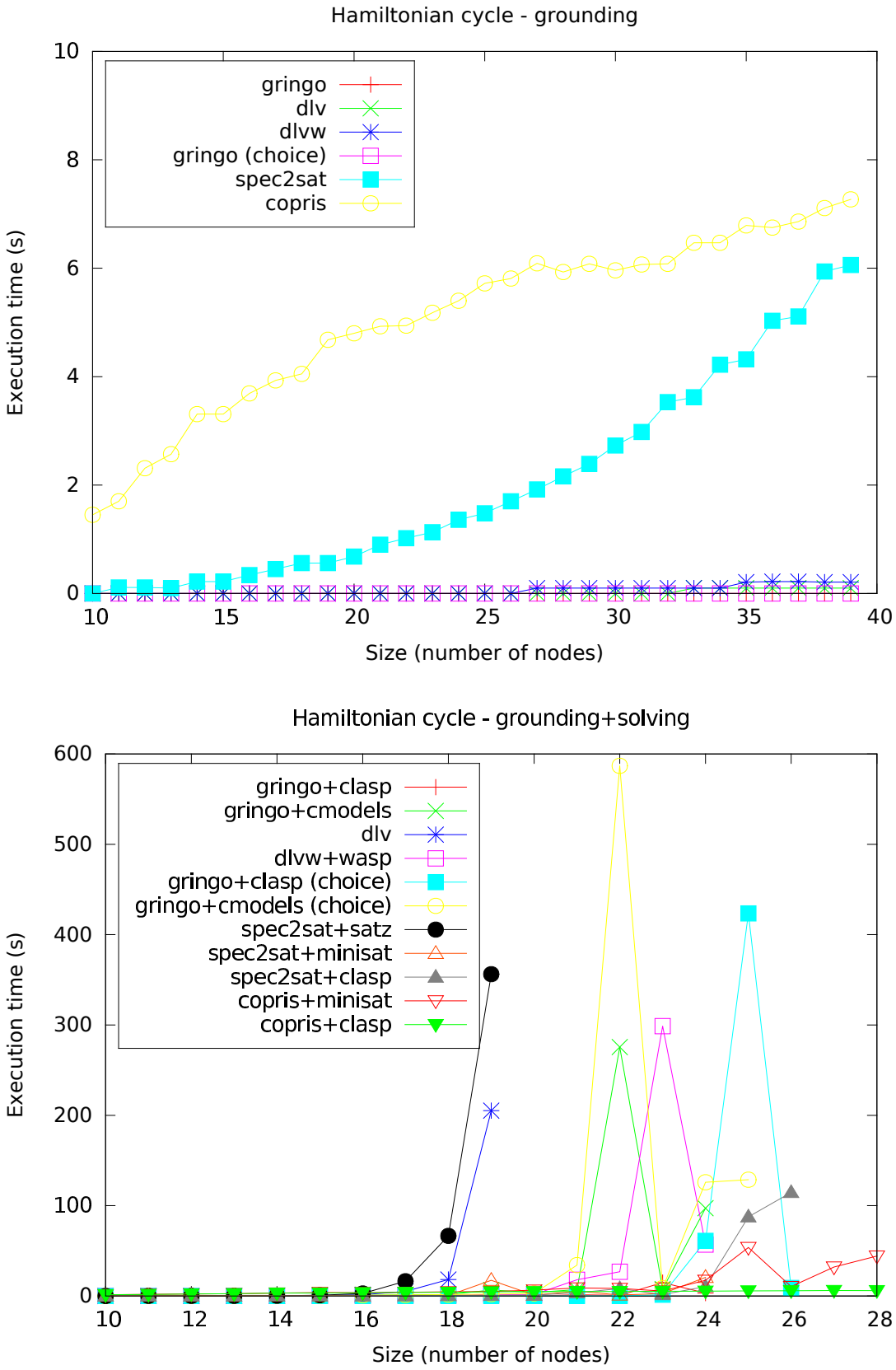


Figure 8. Average running time on Hamiltonian cycle

second.

The last benchmark we considered is Hamiltonian cycle, whose result is reported in Figure 8. We observe that in this case the grounding phase is easy in general, still showing a sensible advantage of ASP grounders over SPEC2SAT. For the solving phase we note that among the systems without learning, DLV performs better than satz, while among the other systems we observe that clasp, cmodels and wasp solved instances up to 24 nodes, and the same holds for minisat on the output of SPEC2SAT. We also remark that clasp was in general faster than cmodels and wasp, and its performance is even better on the rewriting based on choice rules. In fact, on the encoding with choice rules, clasp solved instances up to 26 nodes. A similar result has been obtained by clasp on the output of SPEC2SAT, with running times often smaller than those obtained on the output of NPSPEC2ASP. Concerning copris, we note that it is in general faster than other systems, solving instances up to 28 nodes.

5. Conclusion

In this paper we have presented a transformation of NP-SPEC programs into ASP. The translation is modular and not complex at all, allowing for very efficient transformations. Compared to the previously available transformation into Boolean satisfiability, there are a number of crucial differences: While our transformation is from a formalism with variables into another formalism with variables, Boolean satisfiability of course does not allow for object variables. Therefore any transformation to that language has to do an implicit instantiation. It is obvious that instantiation can be very costly, and thus using sophisticated instantiation methods is usually of utmost importance. However, optimization methods for instantiation are often quite involved and not easy to implement, and therefore adopting them in a transformation is detrimental. After all, the appeal of transformations are usually their simplicity and the possibility to re-use existing software after the transformation.

Our transformation method does just that; by not instantiating it is possible to re-use existing instantiators inside ASP systems, many of which use quite sophisticated techniques like join ordering heuristics, dynamic indexing and many more. We have provided a prototype implementation that showcases this advantage. Already on rather small examples that were used previously for evaluating NP-SPEC implementations a considerable advantage of our method can be observed. This impression is confirmed by more systematic experiments involving domains from ASP competitions reformulated for NP-SPEC. For these domains, we performed a scalability analysis that clearly show a better asymptotic behavior for the tool chain involving ASP than the tool chain involving SAT. This finding is independent of the concrete ASP and SAT systems considered. However, the experiments also clearly highlight performance differences for ASP systems.

There is a second aspect of our work, which regards ASP. As can be seen in Section 3, the translation of $\mathrm{Permutation}$ either gives rise to possibly many integrity constraints or one with an aggregate. In any case, all current ASP instantiators will materialize all associations between tuples of the domain definition and the permutation identifiers, even if the identifiers are not really important for solving the problem. This means that there are obvious symmetries in the instantiated program. There exist proposals for symmetry breaking in ASP (e.g. (Drescher, Tifrea, & Walsh, 2011)), but they typically employ automorphism detection. We argue that in cases like this, a statement like $\mathrm{Permutation}$, $\mathrm{Partition}$, or $\mathrm{IntFunc}$ would make sense as a language addition for ASP solvers, which could exploit the fact that the permutation identifiers introduce a particular known symmetry pattern that does not have to be detected by any external tool.

Acknowledgements

Supported by Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l., and by National Group for Scientific Computation (GNCS-INDAM).

References

Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., ... Xiao, G. (2013). The fourth answer set programming competition: Preliminary report. In P. Cabalar & T. C. Son (Eds.), *12th international conference on logic programming and nonmonotonic reasoning (lpnmr 2013)* (Vol. 8148, pp. 42–53). Springer Berlin/Heidelberg.

Alviano, M., Faber, W., Leone, N., Perri, S., Pfeifer, G., & Terracina, G. (2011). The disjunctive datalog system DLV. In G. Gottlob (Ed.), *Datalog 2.0* (Vol. 6702, pp. 282–301). Springer Berlin/Heidelberg.

Apt, K. R., Blair, H. A., & Walker, A. (1988). Towards a Theory of Declarative Knowledge. In J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming* (pp. 89–148). Washington DC: Morgan Kaufmann Publishers, Inc.

Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Ben-Eliyahu, R., & Dechter, R. (1994). Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, *12*, 53–87.

Cadoli, M., Ianni, G., Palopoli, L., Schaerf, A., & Vasile, D. (2000). An Executable Specification Language for Solving all the Problems in NP. *Computer Languages*, *26*(2/4), 165–195.

Cadoli, M., Mancini, T., & Patrizi, F. (2006). SAT as an effective solving technology for constraint problems. In F. Esposito, Z. W. Ras, D. Malerba, & G. Semeraro (Eds.), *Foundations of intelligent systems, 16th international symposium, ismis 2006, bari, italy, september 27-29, 2006, proceedings* (Vol. 4203, pp. 540–549). Springer.

Cadoli, M., Palopoli, L., Schaerf, A., & Vasile, D. (1999). NP-SPEC: An executable specification language for solving all problems in NP. In *Proceedings of the first international workshop on practical aspects of declarative languages* (Vol. 1551, pp. 16–30). Springer.

Cadoli, M., & Schaerf, A. (2005). Compiling problem specifications into SAT. *Artificial Intelligence*, *162*(1–2), 89–120.

Calimeri, F., Ianni, G., & Ricca, F. (2014). The third open answer set programming competition. *TPLP*, *14*(1), 117–135. Retrieved from `http://dx.doi.org/10.1017/S1471068412000105` doi:

Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., ... Veltri, P. (2011). The third answer set programming competition: Preliminary report of the system competition track. In J. Delgrande & W. Faber (Eds.), *11th international conference on logic programming and nonmonotonic reasoning (lpnmr 2011)* (Vol. 6645, p. 388-403). Springer Berlin/Heidelberg. Retrieved from `http://dx.doi.org/10.1007/978-3-642-20895-9_46`

Dodaro, C., Alviano, M., Faber, W., Leone, N., Ricca, F., & Sirianni, M. (2011). The birth of a WASP: Preliminary report on a new ASP solver. In F. Fioravanti (Ed.), *26th italian conference on computational logic (cilc 2011)* (Vol. 810, pp. 99–113). Sun SITE Central Europe. Retrieved from `http://ceur-ws.org/Vol-810/paper-l06.pdf`

Drescher, C., Tifrea, O., & Walsh, T. (2011). Symmetry-breaking answer set solving. *AI Communications*, *24*(2), 177–194.

Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In *Sat* (pp. 502–518).

Eiter, T., Fink, M., & Woltran, S. (2007). Semantical Characterizations and Complexity of Equivalences in Stable Logic Programming. *ACM Transactions on Computational Logic*, *8*(3), 1–53.

Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., & Ielpa, G. (2008). Design and implementation of aggregate functions in the dlv system. , *8*(5–6), 545–580. doi:

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., & Schneider, M. T. (2011). Potassco: The potsdam answer set solving collection. *AI Communications*, *24*(2), 107–124.

Gebser, M., Kaufmann, B., Neumann, A., & Schaub, T. (2007, January). Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)* (pp. 386–392). Morgan Kaufmann Publishers.

Gebser, M., Schaub, T., & Thiele, S. (2007, May). Gringo : A new grounder for answer set programming. In C. Baral, G. Brewka, & J. Schlipf (Eds.), *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07* (Vol. 4483, pp. 266–271). Tempe, Arizona: Springer Verlag. doi:

Gelfond, M., & Lifschitz, V. (1991). Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, *9*, 365–385.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., & Scarcello, F. (2006, July). The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, *7*(3), 499–562.

Li, C. M. (1999). A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, *71*(2), 75–80.

Lierler, Y., & Maratea, M. (2004, January). Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In V. Lifschitz & I. Niemelä (Eds.), *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)* (Vol. 2923, pp. 346–350). Springer.

Mariën, M., Wittocx, J., Denecker, M., & Bruynooghe, M. (2008). SAT(ID): satisfiability of propositional logic extended with inductive definitions. In H. K. Büning & X. Zhao (Eds.), *Theory and applications of satisfiability testing - SAT 2008, 11th international conference, SAT 2008, guangzhou, china, may 12-15, 2008. proceedings* (Vol. 4996, pp. 211–224). Springer. Retrieved from `http://dx.doi.org/10.1007/978-3-540-79719-7_20` doi:

Minker, J. (Ed.). (1988). *Foundations of Deductive Databases and Logic Programming*. Washington DC: Morgan Kaufmann Publishers, Inc.

Pelov, N., & Ternovska, E. (2005). Reducing inductive definitions to propositional satisfiability. In M. Gabbrielli & G. Gupta (Eds.), *Logic programming, 21st international conference, ICLP 2005, sitges, spain, october 2-5, 2005, proceedings* (Vol. 3668, pp. 221–234). Springer. Retrieved from `http://dx.doi.org/10.1007/11562931_18` doi:

Tamura, N., Taga, A., Kitagawa, S., & Banbara, M. (2009). Compiling finite linear csp into sat. *Constraints*, *14*(2), 254-272.

Van Gelder, A. (1988). Negation as Failure Using Tight Derivations for General Logic Programs. In J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming* (pp. 1149–1176). Washington DC: Morgan Kaufmann Publishers, Inc.