Translating NP-SPEC into ASP*

Mario Alviano and Wolfgang Faber

Department of Mathematics
University of Calabria
87036 Rende (CS), Italy
mario@alviano.net, wf@wfaber.com

Abstract. 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 various forms of ASP, and provide an experimental evaluation of existing implementations and the proposed translations to ASP using various ASP solvers. 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 [4, 6] 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". Some practical features are added to this basic language, often by means of reductions.

The original software system supporting NP-SPEC was described in [4] and was written in the ECL^iPS^e Constraint Programming System, based on Prolog. A second software system, SPEC2SAT[1], was proposed in [5], which rewrites NP-SPEC into propositional formulas for testing satisfiability. The system has also been tested quite extensively in [7], 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

* A preliminary version of this work has been presented at ASPOCP 2012.
Proceedings of the 20th RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2013).
Rome, Italy, June 14–15, 2013.

[1] http://www.dis.uniroma1.it/cadoli/research/projects/NP-SPEC/code/SPEC2SAT/

NP-SPEC problem descriptions are in general not propositional, and therefore a reduction from NP-SPEC to SAT has to include an implicit 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 various variants of ASP. We discuss properties and limitations of the translation and also provide a prototype implementation, for which we provide a preliminary experimental analysis, which shows that our approach is advantageous, in particular that it pays off if grounding tasks are delegated to existing systems. The rest 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 and limitations. In section 4 we report on preliminary experimental results. Finally, in section 5 we draw our conclusions.

2 Preliminaries: NP-SPEC and ASP

2.1 NP-SPEC

We first provide a brief definition of NP-SPEC programs. For details, we refer to [4]. We also note that a few minor details in the input language of SPEC2SAT (in which the publicly available examples are written) are different to what is described in [4]. We will usually stick to the syntax of SPEC2SAT.

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.

`DATABASE`**.** The database section defines extensional predicates or relations and (interpreted) constants. Extensional predicates are defined by writing $\mathtt{p} = \{\mathtt{t}_1, \ldots, \mathtt{t}_n\}$; where $\mathtt{p}$ is a predicate symbol and each $\mathtt{t}_i$ 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 round brackets. Unary extensions that are ranges of integers can also be abbreviated to $\mathtt{n..m}$, where $\mathtt{n}$ and $\mathtt{m}$ are integers or interpreted constants. Constant definitions are written as $\mathtt{c} = \mathtt{i}$; where $\mathtt{i}$ 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.

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

[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.

`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* `Subset(d, p)`, `Permutation(d, p)`, `Partition(d, p, n)`, and `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 [4]. Here, `d` is a *domain definition*, which is either an extensional predicate, a range `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 `subgraph`.

```
SPECIFICATION
  Subset(edge, subgraph).
```

`Permutation(d, p)`. Concerning this construct, `d` is again a domain definition, and `p` will have an extension in which each tuple of `d` is present and an additional argument associates a unique integer between 1 and the cardinality of the extension of `d` (say, `c`) to each tuple, thereby defining a permutation. The extensions of `p` thus define a bijective functions from tuples of the extension 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 $\{1..n\}$, 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

$$\begin{array}{l}\texttt{partition}(1,2,0),\ \texttt{partition}(3,1,0),\ \texttt{partition}(2,3,0),\\ \texttt{partition}(6,2,0),\ \texttt{partition}(5,6,1),\ \texttt{partition}(4,5,1),\\ \texttt{partition}(3,5,1),\ \texttt{partition}(1,4,1),\ \texttt{partition}(4,1,1).\end{array}$$

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

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

$$\begin{array}{l}\texttt{SPECIFICATION}\\ \quad \texttt{IntFunc}(\texttt{edge},\texttt{partition},0..1).\end{array}$$

Stratified Datalog Program. The stratified Datalog program is written using $<--$ as the rule implication symbol. It may contain built-in predicates ($==$, $<$, $>$, $>=$, $<=$, $!=$), arithmetic expressions, and stratified aggregates (COUNT, SUM, MIN, MAX). It may also contain integrity constraints, in which case rule heads contain the special symbol fail. Rule implication is denoted by $<--$, the aggregates are written as for example $\texttt{SUM}(\texttt{p}(*,\_,\texttt{Y}),\texttt{Z}:\texttt{n..m})$ 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 n..m. Comments may be 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:

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

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 by means of Datalog with Circumscription, in which some predicates are minimized. That means that among all

models only those which are minimal with respect to the minimized predicates are accepted. Moreover, among these only those which make the special symbol `fail` false are considered and referred to as answers. All metafacts are reduced to the basic metafact `Subset` that effectively states that the predicate defined by the metafact is not minimized. For further details of the semantics, we refer to [4].

2.2 ASP

Concerning ASP, we only give a very brief overview, details may be found in works such as [3, 12, 16]. An ASP program consists of rules

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

where the $\mathtt{L_i}$ 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 [15] and also guarantees minimality of the answer sets.

Practical ASP systems differ in several details, for instance several do not support disjunction in rule heads, built-in predicates and arithmetic expressions may differ and also aggregates are sometimes written in slightly different ways. In this paper, we will use the syntax of gringo 3 (`http://potassco.sourceforge.net/`) and DLV (`http://www.dlvsystem.com`). Both systems assume that the input programs are safe, that is, each variable in a rule must also occur in a positive body atom. While gringo can also parse disjunctive programs, clasp, the solver it is often used with, can only deal with nondisjunctive programs.

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:

```
#const n = 6
edge(1,2). edge(3,1). edge(2,3). edge(6,2). edge(5,6).
edge(4,5). edge(3,5). edge(1,4). edge(4,1).
d(1..n).
path(X,1)∨path(X,2)∨path(X,3)∨path(X,4)∨path(X,5)∨path(X,6) :- d(X).
:- path(X,A), path(Y,A), X != Y.
:- path(X,P), path(Y,Z), not edge(X,Y), Z = P + 1.
:- path(X,n), path(Y,1), not edge(X,Y).
```

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

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

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 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 $\mathtt{p} = \{\mathtt{t}_1, \ldots, \mathtt{t}_n\}$ will be translated to facts $\mathtt{p}(\mathtt{t}_1) \cdots \mathtt{p}(\mathtt{t}_n)$, and one of the form $\mathtt{p} = \{\mathtt{n}..\mathtt{m}\}$ will be translated to facts $\mathtt{p}(\mathtt{n}) \cdots \mathtt{p}(\mathtt{m})$. Constant declarations such as $\mathtt{c} = \mathtt{i}$, 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 $\mathtt{p} >< \mathtt{q}$, the following set of facts is created: $\{\mathtt{d}(\mathtt{x}_1, \ldots, \mathtt{x}_{i+j}) \mid \mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_i) \wedge \mathtt{q}(\mathtt{x}_{i+1}, \ldots, \mathtt{x}_{i+j})\}$, where `i` and `j` are the arities of `p` and `q`, respectively;
- for the union $\mathtt{p} + \mathtt{q}$, the following set of facts is created: $\{\mathtt{d}(\mathtt{x}_1, \ldots, \mathtt{x}_i) \mid \mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_i) \vee \mathtt{q}(\mathtt{x}_1, \ldots, \mathtt{x}_i)\}$, where `i` is the arity of both `p` and `q`;
- for the intersection $\mathtt{p} * \mathtt{q}$, the following set of facts is created: $\{\mathtt{d}(\mathtt{x}_1, \ldots, \mathtt{x}_i) \mid \mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_i) \wedge \mathtt{q}(\mathtt{x}_1, \ldots, \mathtt{x}_i)\}$, where `i` is the arity of both `p` and `q`; and
- for the difference $\mathtt{p} - \mathtt{q}$, the following set of facts is created: $\{\mathtt{d}(\mathtt{x}_1, \ldots, \mathtt{x}_i) \mid \mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_i) \wedge \neg.\mathtt{q}(\mathtt{x}_1, \ldots, \mathtt{x}_i)\}$, where `i` is the arity of both `p` and `q`, and $\neg.\mathtt{q}(\mathtt{x}_1, \ldots, \mathtt{x}_i)$ is true if and only if the fact $\mathtt{q}(\mathtt{x}_1, \ldots, \mathtt{x}_i)$ 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 $\mathtt{Subset}(\mathtt{d}, \mathtt{p})$, for which we produce

$$\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_n) \vee -\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_n) \ :- \ \mathtt{d}(\mathtt{X}_1, \ldots, \mathtt{X}_n).$$

If available (for instance when using gringo or lparse), we can also use choice rules for translating $\mathtt{Subset}(\mathtt{d}, \mathtt{p})$:

$$\{\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_n) : \mathtt{d}(\mathtt{X}_1, \ldots, \mathtt{X}_n)\}.$$

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

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

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 $\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,1)\cdots\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,c)$ 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. As an alternative to the disjunctive rule, one can use a choice rule

$$1\{\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,1..c)\}1 :- \mathtt{d}(\mathtt{X}_1,\ldots,\mathtt{X}_n).$$

Instead of the n integrity constraints it is possible to write just one using an aggregate, if available. In the DLV syntax, one could write

$$:- \#\mathtt{count}\{\mathtt{X}_1,\ldots,\mathtt{X}_n : \mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{A})\} > 1,\ \mathtt{p}(\_,\ldots,\_,\mathtt{A}).$$

or in gringo syntax

$$:- 2\ \#\mathtt{count}\{\mathtt{p}(\mathtt{X}1,\ldots,\mathtt{X}_n,\mathtt{A})\},\ \mathtt{p}(\_,\ldots,\_,\mathtt{A}).$$

The remaining metafacts are actually much simpler to translate, as the bijection criterion does not have to be checked. The following table shows the translations, where n is the arity of d, and DLV syntax is listed above, gringo syntax below.

$\mathtt{Partition}(\mathtt{d},\mathtt{p},\mathtt{k})$	$\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,0)\vee\ldots\vee\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{k}-1)\ :-\ \mathtt{d}(\mathtt{X}_1,\ldots,\mathtt{X}_n).$
	$1\{\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,0..\mathtt{k}-1)\}1 :- \mathtt{d}(\mathtt{X}_1,\ldots,\mathtt{X}_n).$
$\mathtt{IntFunc}(\mathtt{d},\mathtt{p},\mathtt{i..j})$	$\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{i})\vee\ldots\vee\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{j})\ :-\ \mathtt{d}(\mathtt{X}_1,\ldots,\mathtt{X}_n).$
	$1\{\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{i..j})\}1 :- \mathtt{d}(\mathtt{X}_1,\ldots,\mathtt{X}_n).$

What remains are the Datalog rules of the SPECIFICATION section. Essentially, each $\mathtt{Head} <\!\!- \mathtt{Body}$ is directly translated into $\mathtt{Head}' :- \mathtt{Body}'$, with only minor differences. If Head is fail, then $\mathtt{Head}'$ is empty, otherwise it will be exactly the same. The difference between Body and $\mathtt{Body}'$ 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 (#abs instead of abs, #pow instead of ^), while DLV is much more restrictive. DLV currently does not support negative integers and it does not provide constructs corresponding to ^. 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 $\mathtt{SUM}(\mathtt{p}(*,\_,\mathtt{Y}),\mathtt{Z}:\mathtt{n..m})$ is written in DLV's and gringo's syntax, respectively, as

$$\#\mathtt{sum}\{\mathtt{X} : \mathtt{p}(\mathtt{X},\_,\mathtt{Y})\} = \mathtt{Z},\ \mathtt{d}(\mathtt{Z}) \qquad \mathtt{Z}\ \#\mathtt{sum}[\mathtt{p}(\mathtt{X},\_,\mathtt{Y}) = \mathtt{X}]\ \mathtt{Z},\ \mathtt{d}(\mathtt{Z})$$

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

$$\#\mathtt{count}\{\mathtt{X}_1,\mathtt{X}_2 : \mathtt{p}(\mathtt{X}_1,\_,\mathtt{X}_2,\mathtt{Y})\} = \mathtt{Z},\ \mathtt{d}(\mathtt{Z}) \qquad \mathtt{Z}\ \#\mathtt{count}\{\mathtt{p}(\mathtt{X}_1,\_,\mathtt{X}_2,\mathtt{Y})\}\ \mathtt{Z},\ \mathtt{d}(\mathtt{Z}).$$

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.

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 at the moment does only rudimentary correctness checks of the program and is focused on generating ASP programs for correct NP-SPEC input. Moreover, at the moment it generates only the disjunctive rules described in section 3 rather than the choice rules, but we plan to add the possibility to create variants of the ASP code in the near future. For the experiments, the transformation used for `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 in ASP with respect to the previously available transformation into SAT. 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. Below we provide some more details on the *miscellanea* and *csplib2npspec* benchmarks.

Coloring is an instance of the *Graph Coloring* problem, i.e., given a graph G and a set of k colors, checking whether it is possible to assign a color to each node of G in such a way that no adjacent nodes of G share the same color. In the *Diophantine* problem, three positive integers a, b, c are given, and an integer solution to the equation $ax^2 + by = c$ is asked for. The *Factoring* problem consists of finding two non-trivial factors (i.e., greater than 1) of a given integer n. In the *Hamiltonian Cycle* problem a graph G is given, and a cycle traversing each node exactly once is searched. An instance of the *Job Shop Scheduling* problem consists of integers n (jobs), m (tasks), p (processors), and D (global deadline). Jobs are ordered collections of tasks, and each task is performed on a processor for some time. Each processor can perform one task at a time, and the tasks belonging to the same job must be performed in order. The problem is checking whether it is possible for all jobs to meet deadline D. In the *Protein Folding* problem, a sequence of n elements in $\{H, P\}$ is given, and the goal is to find a connected, non-overlapping shape of the sequence on a bi-dimensional, discrete grid, so that the number of "contacts", i.e., the number of non-sequential pairs of H for which the Euclidean distance of the positions is 1, is in a given range R. In the *Queens* problem, an integer n is given, and the goal is to place n non-attacking queens on a $n \times n$ chessboard. In the tested instance, $n = 5$. Given an array A of integers, the *Sorting* problem consists of arranging the elements of A in non-descending order. An instance

of the *Subset Sum* problem comprises a finite set A, a size $s(a) \in \mathbb{N}^+$ for each $a \in A$, and $B \in \mathbb{N}^+$. The goal of the problem is checking whether there is a subset A' of A such that the sum of the sizes of the elements in A' is exactly B. In a $Sudoku$, the goal is to fill a given (partially filled) grid with the numbers 1 to 9, so that every column, row, and 3×3 box indicated by slightly heavier lines has the numbers 1 to 9. *3-SAT* is a well-known NP-complete problem: Given a propositional formula T in conjunctive normal form, in which each clause has exactly three literals, is T satisfiable, i.e., does there exist an assignment of variables of T to $\{true, false\}$ that makes T evaluate to $true$? The *Tournament Scheduling* problem consists of assigning the matches to rounds of a round-robin tournament for a sports league. The match is subject to several constraints, such as: (i) complementary teams t_1 and t_2 have complementary schedules, i.e., for each round r, if t_1 plays home in r then t_2 plays away in r, and vice versa; (ii) two top matches cannot take place at distance smaller than a given value; (iii) any team cannot match two top teams at distance smaller than a given value. (See [7] for details.)

Given $n \in \mathbb{N}$, find a vector $s = (s_1, ..., s_n)$ such that (i) s is a permutation of $Z_n = \{0, 1, \ldots, n-1\}$; and (ii) the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, \ldots, |s_n - s_{n-1}|)$ is a permutation of $Z_n \setminus \{0\} = \{1, 2, \ldots, n-1\}$. A vector v satisfying these conditions is called an all-interval series of size n; the problem of finding such a series is the *All-interval Series* problem of size n. In the *BACP* (balanced academic curriculum problem), each course has associated a number of credits and can have other courses as prerequisites. The goal is to assign a period to every course in a way that the number of courses and the amount of credits per period are in given ranges, and the prerequisite relationships are satisfied. A *BIBD* is defined as an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. In the *Car Sequencing* problem, a number of cars are to be produced; they are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sun-roof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. In the testcase there are 10 cars, 6 variants on a basic model, and 5 options. A *Golomb ruler* is a set of m integers $0 = a_1 < a_2 < \cdots < a_m$ such that the $m(m-1)/2$ differences $a_j - a_i$ $(1 \leq i < j \leq m)$ are distinct. *Langford*'s problem is to arrange k sets of numbers 1 to n so that each appearance of the number m is m numbers on from the last. Given integers n and b, the objective of the *Low Autocorrelation* problem is to construct a binary sequence S_i of length n, where each bit takes the value +1 or -1, so that $E = \sum_{k=1}^{n-1} (C_k)^2 \leq b$, where $C_k = \sum_{i=0}^{n-k-1} S_i \cdot S_{i+k}$. An order n *magic square* is a $n \times n$ matrix containing the numbers 1 to n^2, with each row, column and main diagonal summing up to the same value. The *Ramsey* problem is to color the edges of a complete graph with n nodes using at most k colors, in such a way that there is no monochromatic triangle in the graph. The *Round-robin Tournament* problem is to schedule a tournament of n teams over $n-1$ weeks, with each week divided into $n/2$ periods, and each period divided into two slots. A tournament must satisfy the following three constraints: every team plays once a week; every team plays at most twice in the

same period over the tournament; every team plays every other team. *Schur's Lemma* problem is to put n balls labeled $\{1, \ldots, n\}$ into 3 boxes so that for any triple of balls (x, y, z) with $x + y = z$, not all are in the same box. In the *Social Golfer* problem there are n golfers, each of whom play golf once a week, and always in groups of s. The goal is to determine a schedule of play for these golfers, to last l weeks, such that no golfer plays in the same group as any other golfer on more than one occasion.

The experiment was executed on an Intel Core2 Duo P8600 2.4 GHz with 4 GB of central memory, running Linux Mint Debian Edition (wheezy/sid) with kernel Linux 3.2.0-2-amd64. Memory was limited to 3 GB and time to 600 seconds. The tools SPEC2SAT and NPSPEC2ASP are compiled with gcc 4.6.3. The other tools involved in the experiment are satz 215.2 [17], minisat 1.14 [11], gringo 3.0.4 [14], clasp 2.0.6 [13], cmodels 3.83 [18], DLV 2011-12-21 [1], and wasp (version alpha) [9].

In our experiment, 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. 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. 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.

Experimental results concerning the *miscellanea* benchmark 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, sometimes by several orders of magnitude. In fact, SPEC2SAT has to compute a ground SAT instance to pass to a SAT solver, while NPSPEC2ASP outputs a non-ground ASP program. A fairer comparison is obtained by adding to the time taken by NPSPEC2ASP the time required by 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. 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.

Concerning the computation of one solution from each ground specification, all considered SAT and ASP solvers are fast in almost all tests. The only exceptions are satz

for *proteinFolding*, which exceeds the allotted time, and DLV for *jobShopScheduling*, whose execution lasted around 94 seconds. We also note that SAT solves are faster than gringo+cmodels for *factoring*, and that DLV has not been tested on 2 instances containing negative integers, which are not supported by DLV.

Table 1. Running times on the *miscellanea* benchmark

Instance	SPEC2SAT				NPSPEC2ASP						
	only	satz	minisat	clasp	DLV inst	DLV	DLV^w inst	DLV^w +wasp	gringo	gringo +clasp	gringo+ cmodels
coloring	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
diophantine	0.76	0.86	0.80	0.82	0.04	0.03	0.07	0.09	0.02	0.06	0.18
factoring	6.19	10.07	6.54	7.63	0.23	0.43	0.46	0.69	0.17	1.19	15.46
hamiltonianCycle	0.02	0.02	0.02	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00
jobShopScheduling	44.95	46.74	46.15	46.17	1.71	93.52	2.32	4.84	1.02	2.20	6.14
proteinFolding	139.47	>600	151.67	142.83	N/A*	N/A*	N/A*	N/A*	2.63	5.08	10.98
queens	0.02	0.02	0.02	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sorting	0.01	0.02	0.02	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.02
subsetSum	0.08	0.09	0.09	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sudoku	3.15	3.27	3.20	3.21	0.10	0.09	0.20	0.31	0.11	0.17	0.27
threeSat	0.00	0.00	0.00	0.00	N/A*	N/A*	N/A*	N/A*	0.00	0.00	0.00
tournamentScheduling	0.45	0.46	0.46	0.46	0.03	0.02	0.04	0.04	0.01	0.01	0.02

* The instance contains negative integers.

Table 2. Instance sizes of the *miscellanea* benchmark

Instance	SPEC2SAT		NPSPEC2ASP					
			DLV		DLV^w		gringo	
	Clauses	Variables	Rules	Atoms	Rules	Atoms	Rules	Atoms
coloring	45	18	40	31	40	31	58	38
diophantine	14,628	140	9,800	142	9800	142	9,940	145
factoring	123,748	498	61,998	500	61998	500	62,496	503
hamiltonianCycle	348	36	261	99	261	99	291	94
jobShopScheduling	209,495	1,980	156,107	2,052	156287	2232	158,087	2,089
proteinFolding	735,721	669	N/A*	N/A*	N/A*	N/A*	520,107	347
queens	165	25	125	65	125	65	145	61
sorting	427	49	252	126	252	126	294	120
subsetSum	1,418	125	49	54	68	91	100	77
sudoku	33,825	1,458	24,777	2,545	25962	2545	25,263	1,736
threeSat	30	39	N/A*	N/A*	N/A*	N/A*	87	76
tournamentScheduling	1,641	108	1,675	115	1793	227	1,810	182

* The instance contains negative integers.

Table 3 reports experimental results concerning the *csplib2npspec* benchmark. We start by observing that instances in this benchmark are more resource demanding than instances in the *miscellanea* benchmark. In fact, we note that *golombRuler* is too difficult for SPEC2SAT, which did not terminate on the allotted time on this instance. On the other hand, the rewriting provided by NPSPEC2ASP is processed in around 39 seconds by gringo+cmodels, in around 30 seconds by gringo+clasp and DLV^w+wasp, and in around 28 seconds by DLV. Another 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 BerkMin heuristics adopted by minisat, clasp and cmodels. In almost all other instances the ASP solvers compute solutions in less than 1 second, while SAT solvers 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 SAT solvers.

The experimental results show that translating NP-SPEC programs into ASP rather than SAT seems to be preferable, due to the fact that sophisticated instantiation techniques can be leveraged. Moreover, also the nondeterministic search components of

Table 3. Running times on the *csplib2npspec* benchmark

Instance	SPEC2SAT				NPSPEC2ASP						
	only	satz	minisat	clasp	DLV inst	DLV	DLV^w inst	DLV^w +wasp	gringo	gringo +clasp	gringo+ cmodels
allInterval	1.48	38.33	>600	>600	0.06	0.98	0.11	0.16	0.05	>600	>600
bacp	6.77	6.55	6.45	6.49	0.00	0.00	0.00	0.01	0.00	0.00	0.04
bibd	4.10	4.36	4.24	4.26	0.02	0.09	0.05	0.07	0.01	0.02	0.42
carSequencing	9.03	15.21	9.18	9.30	0.87	0.84	1.14	1.30	0.32	0.51	1.63
golombRuler	>600	>600	>600	>600	24.85	24.23	28.26	31.09	26.75	30.37	39.51
langford	11.82	13.24	12.88	12.85	0.03	0.89	0.07	0.44	0.02	0.07	22.44
lowAutocorrelation	23.48	24.82	24.11	24.53	N/A*	N/A*	N/A*	N/A*	0.02	0.02	0.03
magicSquare	10.78	11.07	10.92	10.93	0.16	22.50	0.22	1.80	0.11	0.34	>600
ramseyProblem	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
roundrobinTournament	2.32	2.55	2.19	2.19	0.00	0.00	0.01	0.02	0.00	0.00	0.01
schursLemma	0.12	0.12	0.12	0.12	0.00	0.00	0.00	0.00	0.00	0.00	0.00
socialGolfer	7.55	7.78	7.66	7.68	0.08	0.09	0.15	0.21	0.05	0.06	0.27

* The instance contains negative integers.

ASP systems can compete well with SAT solvers, making the use of ASP solvers very attractive for practical purposes.

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 often crucial. 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. Even if only rather small examples were tested, already in most of those cases a considerable advantage of our method can be observed.

Table 4. Instance sizes of the *csplib2npspec* benchmark

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		653610	96 1,149,561	105
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
ramseyProblem	80	30	60	50	60	50	90	61
roundrobinTournament	9,272	456	1,203	275	1203	355	1,467	400
schursLemma	175	30	155	40	155	40	185	51
socialGolfer	21,600	1,424	11,097	441	11105	561	11,321	442

* The instance contains negative integers.
** The system did not terminate in 30 minutes.

There is a second aspect of our work, which regards ASP. As can be seen in section 3, the translation of `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. [10]), but they typically employ automorphism detection. We argue that in cases like this, a statement like `Permutation`, `Partition`, or `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.

Future work consists of consolidating the prototype software and extending it in several directions. In fact, we intend to investigate the possibility to extend our transformation to work with other languages that are similar to NP-SPEC. Moreover, we want to consider alternative translations into SAT using more efficient structures for encoding cardinality constraints [2]. We also want to extend the experiment, which in this paper comprises only benchmarks and instances available on the website of SPEC2SAT. Instances and benchmarks from the 3rd ASP Competition [8] are good candidates for our future experimentation. Finally, we also intend to explore the possibility and impact of introducing `Permutation`, `Partition`, or `IntFunc` into ASP languages.

References

1. M. Alviano, W. Faber, N. Leone, S. Perri, G. Pfeifer, and G. Terracina. The disjunctive datalog system DLV. In G. Gottlob, editor, *Datalog 2.0*, volume 6702 of *Lecture Notes in Computer Science*, pages 282–301. Springer Berlin/Heidelberg, 2011.
2. R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.
3. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
4. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. An Executable Specification Language for Solving all the Problems in NP. *Computer Languages*, 26(2/4):165–195, 2000.
5. M. Cadoli, T. Mancini, and F. Patrizi. SAT as an effective solving technology for constraint problems. In F. Esposito, Z. W. Ras, D. Malerba, and G. Semeraro, editors, *Foundations of Intelligent Systems, 16th International Symposium, ISMIS 2006, Bari, Italy, September 27-29, 2006, Proceedings*, volume 4203 of *Lecture Notes in Computer Science*, pages 540–549. Springer, 2006.
6. M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. 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*, volume 1551 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1999.
7. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162(1–2):89–120, 2005.
8. F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. C. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In J. Delgrande and W. Faber, editors,

11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), volume 6645 of *Lecture Notes in Computer Science*, pages 388–403. Springer Berlin/Heidelberg, 2011.

9. C. Dodaro, M. Alviano, W. Faber, N. Leone, F. Ricca, and M. Sirianni. The birth of a WASP: Preliminary report on a new ASP solver. In F. Fioravanti, editor, *26th Italian Conference on Computational Logic (CILC 2011)*, volume 810 of *CEUR Workshop Proceedings*, pages 99–113. Sun SITE Central Europe, 2011.
10. C. Drescher, O. Tifrea, and T. Walsh. Symmetry-breaking answer set solving. *AI Communications*, 24(2):177–194, 2011.
11. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.
12. M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. T. Schneider. Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.
13. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 386–392. Morgan Kaufmann Publishers, Jan. 2007.
14. M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In C. Baral, G. Brewka, and J. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271, Tempe, Arizona, May 2007. Springer Verlag.
15. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
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 Transactions on Computational Logic*, 7(3):499–562, July 2006.
17. C. M. Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71(2):75–80, 1999.
18. Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In V. Lifschitz and I. Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, volume 2923 of *LNAI*, pages 346–350. Springer, Jan. 2004.