Intermediate Languages of ASP Systems and Tools

Tomi Janhunen

Helsinki University of Technology
Department of Computer Science and Engineering
P.O. Box 5400, FI-02015 TKK, Finland
`Tomi.Janhunen@tkk.fi`

Abstract. In answer set programming (ASP), a search problem is solved by describing its solutions in the input language of an answer set solver which is then used to compute solutions to the problem. Usually, the problem is converted to an intermediate representation before the actual computation of solutions starts. The current ASP systems employ a number of simplified languages (file formats or like) for this purpose. In this paper, we review a number of intermediate languages and analyse their properties. The goal is to identify best features of such languages to be used as the basis of new designs and thus pave the way for the standardisation of intermediate languages in ASP.

1 Introduction

Answer set programming (ASP) [1–3] is an approach to knowledge representation and reasoning in which a search problem is formalised in a logical language so that the models of the representation, i.e., a *logic program*, capture solutions to the problem. Then the models of the program are computed in terms of a dedicated search engine, hereafter called *an answer set solver*. A general architecture for an ASP system is depicted in Figure 1. A full-fledged ASP system provides a programmer with a rule-based *input language* using which problems are encoded. The front-end of the system consists of a parser for this language and the outcome is an *intermediate representation* of the problem in a simplified language directly supported by the search engine. The search of models, i.e., variable assignments potentially fulfilling additional criteria, is then performed using the respective answer set solver. The architecture described above is simplified in the sense that solvers may carry out optional compilation steps—possibly giving rise to additional intermediate representations of the problem.

The goal of this paper is to analyse such intermediate representations and, in particular, general requirements for languages on which they are based. Some of these languages can be merely viewed as machine-readable *file formats* that are easy to parse by the respective solver. Other intermediate languages still resemble input languages in the sense that they come with a concrete human-readable syntax but strict syntactic restrictions may apply. Drawing the borderline between the two extremes may be difficult though. In what follows, we briefly review a number of solvers from the ASP and related domains and point out some intermediate languages of our interest.

– The SMODELS system [4] has its own internal file format—hereafter referred to as the SMODELS format [5]. The front-end of the system, LPARSE, is responsible for

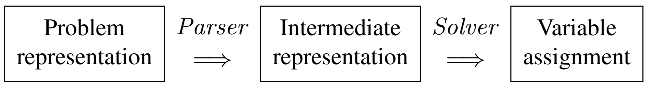

Fig. 1. General Architecture for Answer Set Programming

grounding and partially evaluating the input program which is then passed to the SMODELS engine in the internal format. The user can access this representation but it is not human-readable because of the numerical representation of rules.

– The Center for Discrete Mathematics and Theoretical Computer Science at Rutgers University (DIMACS) has specified two formats for propositional satisfiability problems [6]. The DIMACS/CNF format is the input language for many satisfiability (SAT) solvers[1]. In analogy to the SMODELS format, this format enables the representation of propositional theories in *conjunctive normal form* (CNF).
– A number of ASP systems compile logic programs into propositional theories using Clark's completion procedure [7]. However, additional constraints called *loop formulas* are incrementally introduced to capture answer sets in general. This is the strategy behind the ASSAT [8] and CMODELS [9] systems which understand a subset of the SMODELS format as their input language. The DIMACS/CNF format is used as an intermediate representation for the completion and loop formulas. The author [10] has developed a single-shot transformation for the same purpose. The respective implementation, i.e., the LP2SAT system, supports a subset of the SMODELS format and produces a DIMACS/CNF representation of the program.
– There are mainly two systems developed for disjunctive logic programming: DLV [11] and GNT [12]. As reported by Koch et al. [13], the former system exploits SAT technology in checking the minimality of stable models. This implies that the DIMACS/CNF is used at least indirectly by DLV but the user has no access to the representation. On the other hand, the GNT system consists of two cooperating instances of the SMODELS engine. When GNT is used, disjunctive programs are instantiated using LPARSE and hence an extension of the SMODELS format is used. More recently, the CMODELS system was also extended for proper disjunctive rules.
– *Boolean circuits* (BCs) provide a viable alternative to propositional formulas as they are able to share structure in a very natural way. The BCSAT system [14] implements a check for BC satisfiability and it is based on a file format of its own [15]. The original BCSAT engine solved BCs in this format directly but now an optimised translation into DIMACS/CNF is provided to exploit the rapid improvement of SAT solvers. Therefore we view the BCSAT format as an intermediate language.
– Yet another format has been proposed for pseudo-Boolean solvers which deal with *linear constraints* and *objective functions* rather than plain Boolean constraints. We include the respective input format of PB06 evaluation [16] in our analysis.

In addition to the development of languages and solvers, the ASP community has put forward systematic benchmarking in order to keep track what is the current state of

[1] Many SAT solvers can be accessed through `http://www.satlive.org/`.

	Syntactic expression	**Internal representation**
(1)	$\mathsf{a} \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p, \sim\mathsf{c}_1, \ldots, \sim\mathsf{c}_n.$	$1_\sqcup\#\mathsf{a}_\sqcup(p+n)_\sqcup n_\sqcup$ $\#\mathsf{c}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{c}_{n\sqcup}\#\mathsf{b}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{b}_p\hookleftarrow$
(2)	$\mathsf{a} \leftarrow l\ \{\mathsf{b}_1, \ldots, \mathsf{b}_p, \sim\mathsf{c}_1, \ldots, \sim\mathsf{c}_n\}.$	$2_\sqcup\#\mathsf{a}_\sqcup(p+n)_\sqcup n_\sqcup l_\sqcup$ $\#\mathsf{c}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{c}_{n\sqcup}\#\mathsf{b}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{b}_p\hookleftarrow$
(3)	$\{\mathsf{a}_1, \ldots, \mathsf{a}_h\} \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p, \sim\mathsf{c}_1, \ldots, \sim\mathsf{c}_n.$	$3_\sqcup h_\sqcup\#\mathsf{a}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{a}_{h\sqcup}(p+n)_\sqcup n_\sqcup$ $\#\mathsf{c}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{c}_{n\sqcup}\#\mathsf{b}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{b}_p\hookleftarrow$
(4)	$\mathsf{a} \leftarrow l \leq [\ \mathsf{b}_1 = w_1, \ldots, \mathsf{b}_p = w_p,$ $\sim\mathsf{c}_1 = w_{p+1}, \ldots, \sim\mathsf{c}_n = w_{p+n}].$	$5_\sqcup\#\mathsf{a}_\sqcup l_\sqcup(p+n)_\sqcup n_\sqcup$ $\#\mathsf{c}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{c}_{n\sqcup}\#\mathsf{b}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{b}_{p\sqcup}$ $w_{p+1\sqcup}\ldots{}_\sqcup w_{p+n\sqcup}w_{1\sqcup}\ldots{}_\sqcup w_p\hookleftarrow$
(5)	$\mathsf{minimize}[\ \mathsf{b}_1 = w_1, \ldots, \mathsf{b}_p = w_p,$ $\sim\mathsf{c}_1 = w_{p+1}, \ldots, \sim\mathsf{c}_n = w_{p+n}].$	$6_\sqcup 0_\sqcup(p+n)_\sqcup n_\sqcup$ $\#\mathsf{c}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{c}_{n\sqcup}\#\mathsf{b}_{1\sqcup}\ldots{}_\sqcup\#\mathsf{b}_{m\sqcup}$ $w_{p+1\sqcup}\ldots{}_\sqcup w_{p+n\sqcup}w_{1\sqcup}\ldots{}_\sqcup w_p\hookleftarrow$
(6)	$\mathsf{a}_1, \ldots, \mathsf{a}_n$	$0\hookleftarrow\#\mathsf{a}_{1\sqcup}\mathsf{a}_1\hookleftarrow\ldots\hookleftarrow\#\mathsf{a}_{n\sqcup}\mathsf{a}_n\hookleftarrow 0\hookleftarrow$
(7)	$\mathsf{compute}\{\mathsf{b}_1, \ldots, \mathsf{b}_p, \sim\mathsf{c}_1, \ldots, \sim\mathsf{c}_n\}.$	$\mathtt{B+}\hookleftarrow\#\mathsf{b}_1\hookleftarrow\ldots\hookleftarrow\#\mathsf{b}_p\hookleftarrow 0\hookleftarrow$ $\mathtt{B-}\hookleftarrow\#\mathsf{c}_1\hookleftarrow\ldots\hookleftarrow\#\mathsf{c}_n\hookleftarrow 0\hookleftarrow$
(8)	Trailer when c models are to be computed	$c\hookleftarrow$

Table 1. The internal file format of the SMODELS system

the art in ASP. The Dagstuhl initiative [17] led to the development of a dedicated benchmarking system called ASPARAGUS[2]. Already the first competition showed the need of commonly agreed representations for benchmark problems. As the first step in this direction, a *core language* was drafted by the steering committee of the ASP competition at LPNMR'04 [18]. A variant of the core format, the ground core format (GCORE), has been recently proposed by Namasivayam et al. [19]. It is natural to address the GCORE format in this context due to its potential role in future competitions.

The rest of this paper is organised according to the following plan. In Section 2, we describe some of the formats introduced above in more detail. These pieces of information serve as the basis for the analysis and discussion that follows in Section 3. The interoperability of KR systems and the role of intermediate languages in this respect is addressed in Section 4. Recommendations presented in Section 5 conclude this paper.

2 Examples of Intermediate Languages

This section provides an introduction to a number of intermediate languages. Some of them are merely internal file formats exploited by ASP systems in practise whereas others are of more general syntax and nature—some distinctions in this respects will be made in Section 3. Meanwhile we will describe the details of five intermediate languages, i.e., the SMODELS format, the DIMACS/CNF format, the ground CORE format, the PB06 format, and the BCSAT format. Some extensions to these formats will be discussed, too. Two special symbols, literally "$\sqcup$" for (*white*) *space* and "$\hookleftarrow$" for *newline*, appear in the format descriptions for the sake of concise representation.

[2] The system is installed under `http://asparagus.cs.uni-potsdam.de/`.

	Syntactic expression	**Internal representation**
(1)	Header for n atoms and c clauses	$\texttt{p}\textvisiblespace\texttt{cnf}\textvisiblespace n\textvisiblespace c\hookleftarrow$
(2)	Comments	$\texttt{c}\textvisiblespace\textit{comment}\hookleftarrow$
(3)	$\mathsf{b}_1\lor\ldots\lor\mathsf{b}_p\lor\neg\mathsf{c}_1\lor\ldots\lor\neg\mathsf{c}_n.$	$\#\mathsf{b}_1\textvisiblespace\ldots\textvisiblespace\#\mathsf{b}_p\textvisiblespace$ $-\#\mathsf{c}_1\textvisiblespace\ldots\textvisiblespace-\#\mathsf{c}_n\textvisiblespace 0\hookleftarrow$

Table 2. The DIMACS/CNF format

As suggested by the list above, we begin by describing the SMODELS format that provides an intermediate format for delivering a logic program from the front-end LPARSE to the actual SMODELS engine [4] which implements the search for models. A description of the format is included in the Appendix B of [5] but we present an abridgment in Table 1. A basic assumption is that each ground atom a is assigned a unique number denoted by $\#\mathsf{a}$. The representation of a program starts with a listing of its *basic rules* (1), *constraint rules* (2), *choice rules* (3), *weight rules* (4), and *minimize statements* (5) using the respective representations given in Table 1. Each line starts with a fixed code that identifies the type of the rule in question.[3] For instance, a basic rule $\mathsf{a}\leftarrow\mathsf{b},{\sim}\mathsf{c}$ is represented by a single line "`1 1 2 1 3 2`$\hookleftarrow$"—assuming atom numbers $\#\mathsf{a}=1$, $\#\mathsf{b}=2$, and $\#\mathsf{c}=3$. The next part (6) provides the symbol table for the program, i.e., a mapping from atom numbers back to symbols. Programs may involve *invisible* atoms without a symbolic name. Moreover, *compute statements* (7) may be issued in order to constrain models to be computed by the solver. A summary of this information, i.e., atoms assumed to be *true* and *false*, are listed in separate sections each atom on a line of its own. The representation ends with the number of stable models to be computed (8). All models should be computed if this count is nil.

Compared to the SMODELS format, the DIMACS/CNF format [6] has a much simpler structure as specified in Table 2. The representation of a propositional theory in CNF begins with a header line (1) which nicely enables the solver to allocate appropriate data structures for n atoms and c clauses before reading them in. Any number of comments (2) can be included; also before the header and the representation of clauses (3). Actually, clauses are delimited by `0`s so that grouping to separate lines is not necessary although advisable. Unfortunately, some SAT solvers do not support *empty clauses*, i.e., $p=n=0$ in (3), which is disappointing in view of logical completeness. The simplicity of the format, however, suggests the DIMACS/CNF format as a *machine code* for knowledge representation. This view is present in the design of systems like ASSAT, CMODELS, and LP2SAT that transform programs represented in the SMODELS format into a DIMACS/CNF representation. The result of the transformation is usually more complex/spacious than the original representation which goes back to fact that the expressiveness of rules under stable models strictly exceeds that of clauses [10].

The current extensions that have been proposed to the SMODELS format are listed in Table 3. The first rule type (1) with an *ordered disjunction* in the head [20] is used only internally by LPARSE, i.e., rules of this kind never appear in its output. The integration of proper *disjunctive rules* (2) to the CMODELS system led to the introduction of the code

[3] Code `4` is practically unused although the SMODELS engine still supports it.

	Syntactic expression	**Internal representation**
(1)	$\mathsf{a}_1 \times \ldots \times \mathsf{a}_h \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p, {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n.$	$7_\sqcup h_\sqcup \#\mathsf{a}_{1\sqcup} \ldots {}_\sqcup\#\mathsf{a}_{h\sqcup}(p+n)_\sqcup n_\sqcup$ $\#\mathsf{c}_{1\sqcup} \ldots {}_\sqcup\#\mathsf{c}_{n\sqcup}\#\mathsf{b}_{1\sqcup} \ldots {}_\sqcup\#\mathsf{b}_p\hookleftarrow$
(2)	$\mathsf{a}_1 \vert \ldots \vert \mathsf{a}_h \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p, {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n.$	$8_\sqcup h_\sqcup \#\mathsf{a}_{1\sqcup} \ldots {}_\sqcup\#\mathsf{a}_{h\sqcup}(p+n)_\sqcup n_\sqcup$ $\#\mathsf{c}_{1\sqcup} \ldots {}_\sqcup\#\mathsf{c}_{n\sqcup}\#\mathsf{b}_{1\sqcup} \ldots {}_\sqcup\#\mathsf{b}_p\hookleftarrow$
(3)	$\mathsf{b}_1 \vee \ldots \vee \mathsf{b}_p \vee \neg\mathsf{c}_1 \vee \ldots \vee \neg\mathsf{c}_n.$	$9_\sqcup(p+n)_\sqcup n_\sqcup$ $\#\mathsf{c}_{1\sqcup} \ldots {}_\sqcup\#\mathsf{c}_{n\sqcup}\#\mathsf{b}_{1\sqcup} \ldots {}_\sqcup\#\mathsf{b}_p\hookleftarrow$

Table 3. Some extensions to the SMODELS format

8 for such rules. As a result, the new versions of LPARSE are incompatible with the GNT system [12] which abuses choice rules, represented under code 3, as substitutes for disjunctive ones. The plan is to remove this discrepancy in the future versions of GNT. Note that CMODELS is able to handle programs that contain both choice rules and disjunctive rules. The third extension (3) has arisen in the context of translating logic programs into clauses. The idea is to enrich the SMODELS format by incorporating DIMACS/CNF as its subformat. Then tools like LP2SAT can handle rules and clauses on equal basis and form mixed representations of such expressions if appropriate. The status of the extensions listed in Table 3 is still unofficial and their existence in the future is highly dependent on the developers of the tools involved. For now, there is no official body that would control the evolution of the SMODELS format.

The CORE format [18], as decided by the steering committee of the ASP system contest, aims to define a common syntax for disjunctive rules of the form (2) in Table 3.[4] To this end, the format specifies (i) what kind of identifiers are used for constant, variable, and predicate symbols, (ii) the syntax of atomic formulas, (iii) symbols for logical connectives, and finally (iv) the syntax of rules. As an extensive example, the reader may consider a disjunctive rule

[4] Note that basic/normal rules (1) from Table 1 form a special case of such rules.

	Syntactic expression	**Internal representation**
(1)	$\mathsf{a} \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p, {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n.$	$\mathtt{v\#a}_\sqcup\mathtt{:-}_\sqcup\mathtt{v\#b}_1\mathtt{,}_\sqcup\ldots\mathtt{,}_\sqcup\mathtt{v\#b}_p\mathtt{,}_\sqcup$ $\mathtt{not\ v\#c}_1\mathtt{,}_\sqcup\ldots\mathtt{,}_\sqcup\mathtt{not\ v\#c}_n\mathtt{.}\hookleftarrow$
(2)	$l\{\mathsf{a}_1, \ldots, \mathsf{a}_h\}u \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p,$ ${\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n.$	$l_\sqcup\mathtt{\{}_\sqcup\mathtt{v\#a}_1\mathtt{,\ \ldots,\ v\#a}_h\mathtt{\}}_\sqcup u_\sqcup\mathtt{:-}_\sqcup$ $\mathtt{v\#b}_1\mathtt{,}_\sqcup\ldots\mathtt{,}_\sqcup\mathtt{v\#b}_p\mathtt{,}_\sqcup$ $\mathtt{not\ v\#c}_1\mathtt{,}_\sqcup\ldots\mathtt{,}_\sqcup\mathtt{not\ v\#c}_n\mathtt{.}\hookleftarrow$
(3)	$\mathsf{a} \leftarrow l\{\mathsf{b}_1, \ldots, \mathsf{b}_p, {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n\}u.$	$\mathtt{v\#a}_\sqcup\mathtt{:-}_\sqcup l_\sqcup\mathtt{\{}_\sqcup\mathtt{v\#b}_1\mathtt{,}_\sqcup\ldots\mathtt{,}_\sqcup\mathtt{v\#b}_p\mathtt{,}_\sqcup$ $\mathtt{not\ v\#c}_1\mathtt{,}_\sqcup\ldots\mathtt{,}_\sqcup\mathtt{not\ v\#c}_{n\sqcup}\mathtt{\}}_\sqcup u\mathtt{.}\hookleftarrow$
(4)	$\mathsf{a} \leftarrow l \leq$ $[\mathsf{b}_1 = w_1, \ldots, \mathsf{b}_p = w_p,$ ${\sim}\mathsf{c}_1 = w_{p+1}, \ldots, {\sim}\mathsf{c}_n = w_{p+n}]$ $\leq u.$	$\mathtt{v\#a}_\sqcup\mathtt{:-}_\sqcup l_\sqcup\mathtt{\{}_\sqcup$ $\mathtt{v\#b}_1\mathtt{=}w_1\mathtt{,}_\sqcup\ldots\mathtt{,}_\sqcup\mathtt{v\#b}_p\mathtt{=}w_p\mathtt{,}_\sqcup$ $\mathtt{not\ v\#c}_1\mathtt{=}w_{p+1}\mathtt{,}_\sqcup\ldots\mathtt{,}_\sqcup\mathtt{not\ v\#c}_n\mathtt{=}w_{p+n\sqcup}$ $\mathtt{\}}_\sqcup u\mathtt{.}\hookleftarrow$

Table 4. Examples of the GCORE format

	Syntactic expression	**Internal representation**
(1)	Header for v variables and c constraints	$\star\textvisiblespace\texttt{\#variable=}\textvisiblespace v\ \textvisiblespace\texttt{\#constraint=}\textvisiblespace c\hookleftarrow$
(2)	Comments	$\star\textvisiblespace\textit{comment}\hookleftarrow$
(3)	Objective function $a_1v_1+\ldots+a_nv_n$	$\texttt{min:}\textvisiblespace a_1\textvisiblespace\texttt{x}\#v_1\textvisiblespace\ldots\textvisiblespace a_n\textvisiblespace\texttt{x}\#v_n\ \texttt{;}\hookleftarrow$
(4)	$a_1v_1+\ldots+a_nv_n = b$	$a_1\textvisiblespace\texttt{x}\#v_1\textvisiblespace\ldots\textvisiblespace a_n\textvisiblespace\texttt{x}\#v_n\textvisiblespace\texttt{=}\textvisiblespace b\texttt{;}\hookleftarrow$
(5)	$a_1v_1+\ldots+a_nv_n \geq b$	$a_1\textvisiblespace\texttt{x}\#v_1\textvisiblespace\ldots\textvisiblespace a_n\textvisiblespace\texttt{x}\#v_n\textvisiblespace\texttt{>=}\textvisiblespace b\texttt{;}\hookleftarrow$

Table 5. The pseudo-Boolean format used at PB06 competition

```
open(X,Y); closed(X,Y) :- abscissa(X), ordinate(Y).
```

expressed in the CORE syntax. It may be questionable to view this format as an *intermediate* language in the first place because it is merely a specification of a common input language for a number of ASP solvers: A practicality when organising an ASP solver contest. However, the GCORE format [19] is somewhat closer to the SMODELS format in the sense that all rules are assumed to be *ground*. As an indication of this, atom names are substituted by standard names of the form vn where n is a number. Table 4 collects the representations of rules involved in the SMODELS format (recall Table 1) expressed using the GCORE format. Generally speaking, the GCORE format admits a more liberal use of cardinality and weight constraints, recall the bodies of rules (3) and (4) in Table 4, respectively, used in the heads and bodies of rules such as

```
1 {v1, v2} 2 :- 1 {v3, v4}, {v5, v6} 2.
```

In this sense, the format is more general than the SMODELS format, has features of the input language of LPARSE but only ground rules are supported. In view of the original CORE format, however, no representation is reserved for proper disjunctive rules.

The last two formats taken into consideration originate from other paradigms than ASP. Pseudo-Boolean solving generalises satisfiability checking in terms of traditional *linear constraints* and an objective function subject to minimisation. Problems of this kind are represented in a format described in Table 5. The headers (1) and comments (2) are analogous to the DIMACS/CNF format. Boolean variables are represented as in the GCORE format but canonical names start with "x" rather than "v". The first non-comment line may include an objective function (3) to be minimised. The pseudo-Boolean constraints, i.e., equalities (4) and inequalities (5), follow. These constructs resemble weight constraints used in ASP and an objective function can be expressed using a minimisation statement (5) from Table 1. It is good to point out that the PB06 format can be viewed as a generalisation of the DIMACS/CNF format because a traditional Boolean clause $\mathsf{b}_1\lor\ldots\lor\mathsf{b}_p\lor\neg\mathsf{c}_1\lor\ldots\lor\neg\mathsf{c}_n$ can be captured with an inequality $\mathsf{b}_1+\ldots+\mathsf{b}_p-\mathsf{c}_1\ldots-\mathsf{c}_n\geq 1-n$ where b_i's and c_j's take either 0 or 1 as their values.

Boolean circuits provide yet another representation for Boolean functions. The input language of the BCSAT system provides a flexible representation of Boolean circuits in terms of *gate definitions* of the form $g\texttt{:=}f$ where g is the name of the gate and f is a Boolean formula associated with g. The syntax of formulas is summarised in Table 6. Together, a set of gate definitions should form a non-circular definition of the Boolean circuit under consideration. Besides variables, Boolean constants, and standard Boolean

variable | `T` | `F`
F_1 `==` F_2 | `EQUIV(` F_1`,` ... `,`F_n `)`
F_1 `=>` F_2 | `IMPLY(` F_1 `,` F_2 `)`
F_1 `|` F_2 | `OR(` F_1`,` ... `,`F_n `)`
F_1 `&` F_2 | `AND(` F_1`,` ... `,`F_n `)`
`~` F_1 | `NOT(` F_1 `)`
F_1 `^` F_2 | `ODD(` F_1`,` ... `,`F_n `)`
`EVEN(` F_1`,` ... `,`F_n `)`
`ITE(` F_1 `,` F_2 `,` F_3 `)`
`[` l `,` u `] (` F_1`,` ... `,`F_n `)`
`(` F_1 `)`

Table 6. Syntax of formulas used in the BCSAT format

connectives there are primitives for parity checking, the if-then-else connective adopted from *binary decision diagrams* (BDDs), and cardinality constraints. These extensions nicely increase the expressiveness of basic Boolean circuits in view of applications. Gate definitions may be accompanied by *gate assignments* of the forms "`ASSIGN` g`;`" or "`ASSIGN ~`g`;`" which set a specific gate g to true (`T`) or false (`F`), respectively, in analogy to compute statements in the SMODELS format. Finally, we mention that a file in the BCSAT format is supposed to start with a header line "BC1.0↩".

3 Analysis and Discussion

The purpose of this section is to present an analysis of five intermediate languages introduced in Section 2: the DIMACS/CNF, SMODELS, GCORE, PB06, and BCSAT formats. In the sequel, a number of properties of these languages will be pointed out and followed by a discussion on their prevalence among the quintet under consideration. A summary of these results is collected in Table 7. However, certain properties shared/lacked by all formats are not mentioned for space reasons but subsequently discussed in Section 3.1. The labels of the following items refer to the columns of Table 7.

1. **FF**: The language has been designed as a pure (machine-readable) *file format*.
   This is clearly true for DIMACS/CNF and SMODELS formats. As an indication of this, it is straightforward to implement a parser for these formats—a simple automaton will do the job. The ease of parsing is also a goal of the PB06 format although it insists on a support for arbitrarily long integers. A further aspect of the these low-level file formats is that they are no longer valid input for the parser depicted in the general architecture (recall Figure 1), i.e., they do not correspond to a syntactic fragment of the input language. Indeed, the GCORE format is based on a simplified LPARSE syntax in which ground atoms are additionally represented using standard names `v1`, `v2`, ... and so on. This means in principle that programs in the GCORE format can be recycled through the parser but this may not be feasible for the sake of efficiency. For instance, a simplified parser called GLPARSE is exploited by the ASPARAGUS system in order not to affect benchmarking times of solvers by

Format	FF	VI	CL	SN	EX
DIMACS/CNF	×	×	×		
SMODELS	×			×	×
GCORE			×		×
PB06	×		×		×
BCSAT		×	×	×	×

Table 7. Properties of Certain Intermediate Languages Summarised

the time spent on parsing. It is also worth mentioning LPLIST[5] which transforms problem representations in the SMODELS format, or alternatively DIMACS/CNF, back to a symbolic representation that can be parsed again. Among the formats subject to analysis herein, the BCSAT format is in its own category as it is based on a recursive syntax and thus requires more sophisticated methods for parsing. In any case, the BCSAT format needs not be a fragment of the input language of the overall system in analogy to DIMACS/CNF and the SMODELS format.

2. **VI**: The format includes *version information* that enables revisions in the future.
This feature boils down to having a header to carry such information in the format. This is the case for DIMACS/CNF and the BCSAT format although only the latter has a proper version number incorporated. The other three formats do not have headers which makes it difficult to detect format versions reliably. For instance, the extensions of the SMODELS format described in Table 3 cannot be perceived if no rules under codes `7–8` are present. The integrity of headers is naturally a prerequisite for reliable detection. Moreover, it does not appear as a good idea to express version information in comment lines in an ad-hoc manner.
3. **CL**: The use of *comment lines* is allowed.
All formats under consideration except the SMODELS format have this property.
4. **SN**: The language carries *symbolic names* for (propositional) variables.
This property is significant from the user's point of view, i.e., it enables the respective solver to print variable assignments in a human-understandable way in the last phase of answer set computation (recall Figure 1). The users of SAT solvers have to live with the lack of this property in DIMACS/CNF and digest lists of integers or binary vectors in a way or another. Fortunately, the mainstream ASP solvers have carried symbolic information from the very beginning. To this end, the SMODELS format includes a symbol table as specified by (6) in Table 1. On the other hand, the BCSAT format uses symbolic names of variables as lexical elements thus avoiding loss of information in this respect. The lack of support for symbolic names can be alleviated to some extent by incorporating such information within comment lines. But this is only a partial solution because the format itself does not specify the representation of symbolic names which may therefore diverge.
5. **EX**: The language is easily *extendible* with new syntactic expressions.
The poor extendibility of DIMACS/CNF goes back to the type information "`cnf`" given in the header. Thus it is unnatural to introduce new expressions unless several

[5] At least for now, LPLIST is distributed with CIRC2DLP at `http://www.tcs.hut.fi/Software/circ2dlp/`.

representations are concatenated one after another. The flexibility of the SMODELS format has already been demonstrated in Table 3 where new codes for rules are introduced. The encoding of objective functions under the PB06 format (recall Table 5) suggests a strategy for extensions using labels for types. The remaining two formats are easy to extend by new syntax due to flexibility of their grammars.

Interestingly, none of the formats under consideration has all of properties summarised in Table 7. The BCSAT format appears to be closest to having them all. On the other hand, the coverage of syntactic primitives was not introduced as a criterion for the analysis because the languages have been designed for slightly different purposes.

3.1 Further Properties

In what follows, we will address further properties of intermediate languages: (i) pros and cons of binary file formats, (ii) modularity aspects of intermediate languages, and (iii) the possibility of embedding metadata in intermediate representations.

All the formats addressed above are based on a textual (ASCII) representation either using numbers or character strings as lexical tokens. Thus none of them is comparable to *binary file formats* produced by compilers of conventional programming languages. This is perhaps advantageous because, on one hand, binary representations are more tedious to implement in a machine independent way. On the other hand, textual formats provide a less compact representation but can be improved using compression techniques if space complexity becomes a concern.

The study of module systems and modularity in general are receiving growing attention in ASP [21–23]. Inspired by modular notions of program equivalence, the author has implemented a link editor LPCAT[6] for programs in SMODELS format—enabling the construction of larger programs by linking smaller ones together. This is analogous to the use of object modules and libraries in conventional programming systems. For tools like LPCAT symbolic information plays a crucial role and thus formats that support symbolic names are best off in view of implementing a module system. For instance, the SMODELS format does not have a built-in support for modules, i.e., it has been designed in order to represent a single program consisting of a set of rules. However, due to separators used in the SMODELS format, libraries could be formed by simple concatenation of files. Yet another strategy is to use file archive tools for storing program modules, e.g., PKZIP provides random access to compressed modules in contrast to the use of TAR and GZIP. The other format with symbolic names, i.e., the BCSAT format, does neither have a module system. At least in principle, circuit definitions can be joined together as long as the acyclicity of definitions is not endangered by such operations. The headers of circuit descriptions make only a minor obstacle for concatenation.

There are two fundamental pieces of information associated with a symbol: its name and the unique number assigned to it. Invisible symbols, as addressed in [10], can be identified with their numbers. The role of symbols and symbol tables can be developed further in the intermediate languages of ASP systems. Building a proper support for

[6] This tool is used in our translation-based implementation of prioritised circumscription, the PRIO2CIRC system, available at `http://www.tcs.hut.fi/Software/circ2dlp/`.

```
lparse program.lp | smodels
lparse program.lp | lp2atomic | lp2sat | minisat
```

Table 8. Shell pipelines for computing stable models using SMODELS, LP2SAT, and MINISAT

modular program construction requires the distinction of *symbol types* in addition to names. For instance, certain symbols act as the input interface for a module whereas some other symbols mediate its output to other modules in a program. Further extensions become necessary if the support for external function calls is integrated. In the wildest scenarios, we should be ready to associate arbitrary *metadata* with symbols. Such features are not present in the formats listed in Table 7.

4 On the Interoperability of ASP Tools

The development of feasible intermediate languages for ASP solvers can substantially enhance their interoperability and usability with other related tools. So far, our experiences have restricted to the use and development of tools based on the SMODELS format and its extensions as well as DIMACS/CNF. As an example, let us consider the use of LPARSE and SMODELS according to the general ASP architecture in Figure 1. The first line in Table 8 shows an exemplary command line for running LPARSE and SMODELS using a shell pipe "|" in between. When executed, the program in the input file "`program.lp`" is read in and grounded by LPARSE. Then the ground program is forwarded in the SMODELS format through the pipe for the computation of one stable model by SMODELS; further models could be requested using command line options.

The second command line in Table 8 presents a pipeline for the same task but using a translator from the SMODELS format to DIMACS/CNF [10] and the MINISAT solver [24]. The first translator, viz. LP2SAT, removes positive body literals from the program which remains in the SMODELS format. In the next step, another translator called LP2SAT forms the Clark's completion for the program and outputs a DIMACS/CNF representation for it. Finally, MINISAT is used to search a model for the completion. The use of DIMACS/CNF complicates the task of extracting a stable model from the model of the completion because symbolic information is lost in the last phase—decreasing the interoperability of tools involved. However, in order to circumvent this problem in practise, we include a symbol table in the comment lines of the DIMACS/CNF representation and replace MINISAT with a shell script that extracts names of atoms from comments, stores them in a temporary file, runs MINISAT, extracts a model from its output, and maps atom numbers in the model back to their symbolic names. By this procedure we obtain a degree of usability similar to that of SMODELS. These observations suggests a need for an interface specification for ASP/SAT solvers themselves.

In addition to enhanced interoperability, intermediate languages that are commonly agreed upon can facilitate the development of new ASP tools. For instance the rapid advancement of SAT solvers is partly due to a shared format that enables straightforward exchange of benchmarks among the developers. Similar development is going on in the area of ASP. For instance, ASSAT and CMODELS are new solvers that have been developed around the SMODELS format. Quite recently, the combination of LPARSE and SMODELS as illustrated in Table 8 is getting a challenger from that of GRINGO[7] and CLASP [25]. Again, an intermediate format plays a role in this development by separating the phase of parsing and grounding from that of solving and computing models.

In view of the interoperability of systems and tools, it is also worth raising two "political" aspects for discussion. First of all, we have several examples from the software industry where file formats are used as vehicles in marketing policy, i.e., to prevent non-customers from using a particular tool; or to force customers to purchase a new version of the tool for compatibility reasons. To avoid such side-effects in the ASP community, the development of intermediate formats should become a joint standardisation effort the community. The work around the ASP system competition has taken first steps in this direction [18] but this work is still at preliminary stage. In our group, we have taken initiatives in this respect in the development of tools like DLPEQ [26] and CIRC2DLP [27] for disjunctive logic programming. They have been designed to support both GNT [12] and DLV [11] as their back-end solvers as to benefit the users of both systems. The second issue is that new versions of intermediate languages tend to emerge from the initiatives of individual system developers—with little coordination. The same applies to revisions to existing formats which are prone to divergence if there is no control. For instance, the assignment of codes `7–9` in Table 3 is a compromise proposed herein in order to satisfy the needs of a number of tools. A lesson to learn is that, in the long run, the ASP community should have an official body to regulate intermediate languages and to coordinate any proposed extensions to them.

5 Conclusion

In this paper, we have presented the details of five existing intermediate languages related to ASP, brought attention to some of their properties through a systematic analysis, and raised the enhanced interoperability of ASP tools as one of the main goals in the development of new formats. On the basis of the analysis presented in Section 3, my recommendations for any future proposals of intermediate languages are as follows:

1. The format should be easy to extend and for this reason it should also include version information, e.g., for backward compatibility.
2. The format should carry symbolic information; preferably in the form of a symbol table. The entries of the table should have optional fields for type information and metadata, e.g., in view of future extensions.
3. The format should include support for comment lines or a separate section for comments—enabling the integration of documentation in natural language.
4. The format should be based on a textual or numeric representation of the expressions involved in the intermediate language. In comparison with a binary representation, savings in space can still be achieved using explicit compression methods.
5. The format should have a proper module architecture which facilitates modular program development and enables the construction of module libraries.

[7] Available at `http://www.cs.uni-potsdam.de/~sthiele/gringo/`.

The five items above cover most of the aspects raised in the analysis carried out in Section 3. However, one aspect of the format remains open in view of Table 7, i.e., whether to have a numeric low-level file format or one with a more general syntax and symbols as lexical elements. This is somewhat a matter of taste and hence no firm recommendation is spelled out in this respect. It could be even a good idea to have both given translators between the two variants.[8] Nevertheless, the SMODELS and GCORE formats are closest candidates in this respect but as indicated by the recommendations above not totally satisfactory as such—which leaves us with a call for new designs.

It is likely that there are other technical requirements that have not been addressed in this paper and which could serve as a basis for further recommendations. Such factors may also arise in the sequel when ASP evolves as a paradigm. For instance, the support for non-ground representations may become a necessity in the future. There are also other aspects in the development of intermediate formats. Any serious format should be (i) properly published, (ii) provided with basic input/output routines in a number of programming languages, and (iii) equipped with tools, like LPCAT and LPLIST mentioned above, to handle representations in the format. A great deal of organisational work is also required if real *standard formats* are to be developed for the ASP community.

There are also other formats and intermediate languages that can be taken into consideration for the sake of contrast and comparison. In this respect, one candidate is the specification of an on-line library of benchmarks for *satisfiability modulo theories* (SMT-LIB) [28]. However, we excluded the analysis of the SMT-LIB format from the current paper due to its inherent complexity: The format is based on a many-sorted version of first-order logic with equality. In any case, the SMT-LIB format provides an interesting generalisation of propositional theories with external theories and it may provide useful insight how to incorporate external functions and predicates into intermediate languages designed for ASP. In particular, the representation of *aggregates*, such as cardinality and weight constraints introduced above, is of our central interest.

At the moment, the DIMACS/CNF format can be viewed as the de facto standard for representing satisfiability problems for SAT solvers. An interesting question is whether some new intermediate language will reach at least similar status in the ASP community. Hopefully, the five recommendations listed above pave the way in the design of a good candidate for such a language. To this end, it is high time to make serious proposals because expected benefits are manifold. For instance, it is likely that the interoperability of ASP tools is enhanced and the development of ASP solvers is boosted by extensive benchmarking enabled by a standard format. Moreover, a modular format may turn out highly useful in controlling the complexity of grounding which is viewed as a bottleneck of current ASP systems.

Acknowledgements

This work was partially supported by the Academy of Finland under project #211025 titled “Advanced Constraint Programming Techniques for Large Structured Problems”.

[8] Actually, the tools LPLIST and GLPARSE almost provide such facilities for the SMODELS format and the respective fragment of the input language accepted by LPARSE.

References

1. Marek, W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective. Springer-Verlag (1999) 375–398
2. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3–4) (1999) 241–273
3. Gelfond, M., Leone, N.: Logic programming and knowledge representation—the A-Prolog perspective. Artificial Intelligence **138** (2002) 3–38
4. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1–2) (2002) 181–234
5. Syrjänen, T.: Lparse 1.0 user's manual[9]. Available at the SMODELS website (2001) Appendix B, pp. 86–89.
6. DIMACS: Satisfiability suggested format. Available at the ftp server[10] of Rutgers University (1993)
7. Clark, K.L.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press, New York (1978) 293–322
8. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157** (2004) 115–137
9. Lierler, Y., Maratea, M.: CMODELS-2: SAT-based answer set solver enhanced to non-tight programs. [29] 346–350
10. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics **16**(1-2) (2006) 35–86
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
12. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding partiality and disjunctions in stable model semantics. ACM Transactions on Computational Logic **7**(1) (2006) 1–37
13. Koch, C., Leone, N., Pfeifer, G.: Enhancing disjunctive logic programming systems by SAT checkers. Artificial Intelligence **151**(1-2) (2003) 177–212
14. Junttila, T., Niemelä, I.: Towards an efficient tableau method for boolean circuit satisfiability checking. In Lloyd, J.W., et al., eds.: Proc. of CL 2000. Volume 1861 of LNCS., Springer (2000) 553–567
15. Junttila, T.: File format for boolean circuit satisfiability. Available at the BCSAT website[11] (2006)
16. Roussel, O.: PB06: Input format[12]. Available at the PB07 website (2006)
17. Borchert, P., Anger, C., Schaub, T., Truszczyński, M.: Towards systematic benchmarking in answer set programming: The Dagstuhl initiative. [29] 3–7
18. Leone, N., et al.: Core language for ASP solver competitions. Available at the ASPARAGUS website[13] (2004) Minutes of the steering committee meeting at LPNMR'04.
19. Namasivayam, G., Liu, L., Truszczyński, M.: Syntax for ground logic programs – a proposal. Available at URL[14] (2006)

[9] `http://www.tcs.hut.fi/software/smodels/lparse.ps`
[10] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`
[11] `http://www.tcs.hut.fi/~tjunttil/bcsat/`
[12] `http://www.cril.univ-artois.fr/pb07/pb06_inputformat.html`
[13] `http://asparagus.cs.uni-potsdam.de/`
[14] `http://www.cs.uky.edu/ai/groundlp-grammar-proposal.txt`

20. Brewka, G., Niemelä, I., Syrjänen, T.: Implementing ordered disjunction using answer set solvers for normal programs. In Flesca, S., et al., eds.: Proc. of JELIA'02. Volume 2424 of LNCS., Springer (2002) 444–455
21. Ianni, G., Ielpa, G., Pietramala, A., Santoro, A., Calimeri, F.: Enhancing answer set programming with templates. In Delgrande, J.P., Schaub, T., eds.: 10th International Workshop on Non-Monotonic Reasoning, Whistler, Canada, June 6-8, 2004, Proceedings. (2004) 233–239
22. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006). Volume 4079 of LNCS., Springer (2006) 376–390
23. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In Brewka, G., et al., eds.: Proc. of ECAI'06, IOS Press (2006) 412–416
24. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: Proc. of SAT'03. Volume 2919 of LNCS., Springer (2003) 502–518
25. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proc. of IJCAI'07. (2007) 386–392
26. Oikarinen, E., Janhunen, T.: Verifying the equivalence of logic programs in the disjunctive case. [29] 180–193
27. Oikarinen, E., Janhunen, T.: CIRC2DLP—translating circumscription into disjunctive logic programming. In Baral, C., et al., eds.: Proc. of LPNMR'05. Volume 3662 of LNCS., Springer (2005) 405–409
28. Ranise, S., Tinelli, C.: The SMT-LIB standard. Available at the SMT-LIB website[15] (2006) Version 1.2.
29. Lifschitz, V., Niemelä, I., eds.: Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings. In Lifschitz, V., Niemelä, I., eds.: LPNMR. Volume 2923 of LNCS., Springer (2004)

[15] `http://combination.cs.uiowa.edu/smtlib/papers.html`