How to build your own ASP-based system ?!

ROLAND KAMINSKI, JAVIER ROMERO, TORSTEN SCHAUB, and PHILIPP WANKO
University of Potsdam, Germany

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Answer Set Programming, or ASP for short, has become a popular and sophisticated approach to declarative problem solving. Its popularity is due to its attractive modeling-grounding-solving workflow that provides an easy approach to problem solving, even for laypersons outside computer science. However, in contrast to ASP's ease of use, the high degree of sophistication of the underlying technology makes it even hard for ASP experts to put ideas into practice whenever this involves modifying ASP's machinery.

For addressing this issue, this tutorial aims at enabling users to build their own ASP-based systems. More precisely, we show how the ASP system *clingo* can be used for extending ASP and for implementing customized special-purpose systems. To this end, we propose two alternatives. We begin with a traditional AI technique and show how meta programming can be used for extending ASP. This is a rather light approach that relies on *clingo*'s reification feature to use ASP itself for expressing new functionalities. The second part of this tutorial uses traditional programming (in Python) for manipulating *clingo* via its application programming interface. This approach allows for changing and controlling the entire model-ground-solve workflow of ASP. Central to this is *clingo*'s new `Application` class that allows us to draw on *clingo*'s infrastructure by customizing processes similar to the one in *clingo*. For instance, we may apply manipulations to programs' abstract syntax trees, control various forms of multi-shot solving, and set up theory propagators for foreign inferences. A cross-sectional structure, spanning meta as well as application programming, is *clingo*'s intermediate format, *aspif*, that specifies the interface among the underlying grounder and solver. We illustrate the aforementioned concepts and techniques throughout this tutorial by means of examples and several non-trivial case-studies. In particular, we show how *clingo* can be extended by difference constraints and how guess-and-check programming can be implemented with both meta and application programming.

1 Introduction

Answer Set Programming (ASP; Baral 2003; Gebser et al. 2012; Gelfond and Kahl 2014; Lifschitz 2019) has become an established approach to declarative problem solving, experiencing an increasing popularity in academia as well as industry, and sometimes even beyond Artificial Intelligence and Computer Science. This is arguably due to its pursuit of an integrated modeling-grounding-solving paradigm (Gebser and Schaub 2016; Kaufmann et al. 2016) that enables laypersons to use ASP systems off-the-shelf. However, the underlying technology is highly involved and thus much less accessible even for ASP experts. This is also reflected by the fact that there are only two genuine ASP systems nowadays, namely *dlv* (Leone et al. 2006; Alviano et al. 2017) and *clingo* (Gebser et al. 2019), while other computational approaches mostly rely on extensions to these systems or translations into neighboring solving paradigms. This is not without reason and rather due to the high technical sophistication of full-fledged ASP systems. Hence, it is all the more

important to keep this technology open and extensible and so to enable the community to participate in the continuous enhancement of ASP technology. If neglected, we risk a technological gulf that is prone to cut off advances in ASP in the future. Moreover, the extension and integration of ASP technology is indispensable in many real-world applications. Examples include decision support systems for the space shuttle (Nogueira et al. 2001), metabolic network completion (Frioux et al. 2019), and train scheduling (Abels et al. 2021). Hence, empowering the community to master ASP technology also makes it fit for addressing applications at industrial scale.

This empowerment was a guiding motive in the development of the core ASP systems of the *Potsdam Answer Set Solving Collection*, or Potassco for short (Gebser et al. 2011; Gebser et al. 2018), and has meanwhile led to numerous extensions, either being part of Potassco at `potassco.org` or conducted by other scientists worldwide. To further foster such advancements and transfer of ASP technology, we provide in this tutorial an introduction to key techniques allowing advanced users to construct their own ASP systems by building upon Potassco tools.

No matter whether the envisaged system aims at extending ASP or using it as an implementation platform, the key issue is how to capture the added functionality.

To this end, we propose two alternatives.

We begin with a traditional AI technique and show in Section 3 how meta programming can be used for extending ASP. This is clearly the lightest approach in which ASP itself is used to express new functionalities. It draws upon *clingo*'s reification feature for representing the result of grounding a logic program as a set of facts. The original program is then given as data to a meta program that implements the new functionality. In this way, we use *clingo* as a black box and implement all examples by consecutive *clingo* calls. Meta programming is for example used in *asprin* (Brewka et al. 2015) and *plasp* (Dimopoulos et al. 2018).

We then move to the other focus of our tutorial, namely, the use of traditional programming for manipulating *clingo* via its application programming interface (API). This can be seen as treating *clingo* as a gray box, whose modifications are guided through a well-defined interface. Before that, all functionality had to be done by re-programming, a white box approach that needed quite good programming skills. For application interface programming, we have chosen Python as our example language, although other choices exist (e.g. C, C++, Lua, and Rust). This approach allows us to make changes to the entire model-ground-solve workflow of ASP. We detail capabilities and interfaces supporting the implementation of novel ASP technology such as extending the modeling language of *clingo* by means of grammar-based specifications, manipulating the abstract syntax trees of (non-ground) logic programs, as well as multi-shot and theory solving. While multi-shot solving provides us with fine-grained control of ASP reasoning processes, theory solving allows for refining basic ASP solving by incorporating foreign types of constraints. Central to this is *clingo*'s new application class that allows for deriving customized applications from the one of *clingo*. This class constitutes the cornerstone of all recent Potassco systems such as *clingcon* (Banbara et al. 2017), *clingo*[dl] (Janhunen et al. 2017), *eclingo* (Cabalar et al. 2020), and *telingo* (Cabalar et al. 2019). We discuss its role in Section 4 and use it throughout the remaining sections.

Both meta and application interface programming allow for changing the functionality of ASP systems. One difference manifests itself in their degree of elaboration-tolerance (Mc-

Carthy 1998). While a meta encoding benefits from ASP's elaboration-tolerance, this feature is less pronounced in non-declarative programming languages. Here, however, an API makes the difference since it greatly simplifies programming by abstracting from an underlying implementation. Although an API is usually less accessible to ASP users than a meta encoding, it is much easier to handle than any intervention into the programming of the actual ASP system. This differentiation reflects the above distinction of treating an ASP system as a black, gray, or white box, respectively. Also, the possibility of changing an ASP system's functionality brings about the new role of an ASP engineer, which is situated between basic ASP users, using ASP systems as such, and ASP system builders. Obviously, meta programming offers a light entry point for basic users, and its easy accessibility makes it well suited for prototyping new functionalities. On the other hand, API programming can be accomplished with much less programming skills than ASP system building. Also, the usage of an ASP system's API is the predominant use case in industrial applications since it allows for a flexible integration into an existing IT infrastructure. Last but not least, it is instructive to realize that the effectiveness of the chosen alternative depends on the nature of the added functionality. Whenever it can be mapped back onto ASP, meta reasoning might be quite efficient since it harnesses the power of modern ASP systems. Any functionality exceeding ASP's capabilities, however, needs an extension to the system as such.

A cross-sectional structure, spanning meta as well as application interface programming, is *clingo*'s intermediate format, *aspif*, that specifies the interface among the underlying grounder and solver, namely, *gringo* and *clasp*. This is relevant whenever one deals with ground logic programs, be it as reified rules, in machine-readable format, or via the application interface. The whole input, including rules, customized language expressions, as well as all types of directives, is expressed in their ground form in the *aspif* format. The complete *aspif* specification is given in Appendix B. A system that relies on translating ground logic programs in *aspif* format is *lc2casp* (Cabalar et al. 2016).

We illustrate the aforementioned techniques throughout this tutorial by means of examples and several non-trivial case-studies. This includes the computation of classical, supported, here-and-there, and diverse models with meta programming in Section 3, optimization and incremental solving with multi-shot solving in Section 5, hybrid solving and optimization with theory solving in Section 7, and finally guess-and-check programming with both meta and application programming in Sections 3.4 and 8, respectively. The source code of all examples is available online (Potassco Team 2021g; Potassco Team 2021h; Potassco Team 2021d; Potassco Team 2021e).

In what follows, we refrain from distinguishing features of *gringo* and *clasp* and simply refer to them as features of *clingo*. We deal in this tutorial with *clingo* series 5, in particular, *clingo* version 5.5; its installation instructions can be found on its webpage (Potassco Team 2021b). A complete documentation of *clingo*'s API is also available online (Potassco Team 2021c). We rely on a basic familiarity with ASP and its underlying concepts. Comprehensive introductions can be found in several textbooks (Baral 2003; Gebser et al. 2012; Gelfond and Kahl 2014; Lifschitz 2019). Accordingly, we only sketch *clingo*'s input language and refer for details to the *Potassco User Guide* (Gebser et al. 2015). Otherwise, we presuppose some computer science training that permits a basic understanding of shell and Python programming (for meta and application interface programming, respectively).

The core of this tutorial is based on material stemming from an earlier edition on

hybrid answer set solving (Kaminski et al. 2017). The tutorial at hand provides itself the basis of an advanced ASP course, offering complementary teaching material (Potassco Team 2021a).

2 Answer set programming

A logic program consists of rules of the form

$$\texttt{a}_1\texttt{;}\ldots\texttt{;a}_m\ \texttt{:-}\ \texttt{a}_{m+1}\texttt{,}\ldots\texttt{,a}_n\texttt{,not a}_{n+1}\texttt{,}\ldots\texttt{,not a}_o$$

where each $\texttt{a}_i$ is an atom of form $\texttt{p(t}_1\texttt{,}\ldots\texttt{,t}_k\texttt{)}$ and all $\texttt{t}_i$ are terms, composed of function symbols and variables. For $0 \leq m \leq n \leq o$, atoms $\texttt{a}_1$ to $\texttt{a}_m$ are often called head atoms, while $\texttt{a}_{m+1}$ to $\texttt{a}_n$ and $\texttt{not a}_{n+1}$ to $\texttt{not a}_o$ are also referred to as positive and negative body literals, respectively. An expression is said to be ground, if it contains no variables. As usual, `not` denotes (default) negation. A rule is called a fact if $m = n = o = 1$, normal if $m = 1$, and an integrity constraint if $m = 0$. Semantically, a logic program induces a set of stable models, being distinguished models of the program determined by the stable models semantics (Gelfond and Lifschitz 1990).

To ease the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include conditional literals and cardinality constraints (Simons et al. 2002). The former are of the form[1] $\texttt{a:b}_1\texttt{,}\ldots\texttt{,b}_m$, the latter can be written as[2] $\texttt{s}\,\texttt{\{d}_1\texttt{;}\ldots\texttt{;d}_n\texttt{\}}\,\texttt{t}$, where `a` and $\texttt{b}_i$ are possibly default-negated (regular) literals and each $\texttt{d}_j$ is a conditional literal; `s` and `t` provide optional lower and upper bounds on the number of satisfied literals in the cardinality constraint. We refer to $\texttt{b}_1\texttt{,}\ldots\texttt{,b}_m$ as a condition. The practical value of both constructs becomes apparent when used with variables. For instance, a conditional literal like `a(X):b(X)` in a rule's body expands to the conjunction of all instances of `a(X)` for which the corresponding instance of `b(X)` holds. Similarly, `2 {a(X):b(X)} 4` is true whenever at least two and at most four instances of `a(X)` (subject to `b(X)`) are true. More sophisticated examples are given in Section 3, e.g. in Listing 4. Finally, objective functions minimizing the sum over the first argument w_i of a set of weighted tuples $(w_i, \boldsymbol{t}_i)$, whose membership is subject to condition c_i, are expressed as $\texttt{\#minimize\{}w_1\texttt{@}l_1\texttt{,}\boldsymbol{t}_1\texttt{:}c_1\texttt{;}\ldots\texttt{;}w_n\texttt{@}l_n\texttt{,}\boldsymbol{t}_n\texttt{:}c_n\texttt{\}}$. Lexicographically ordered objective functions are (optionally) distinguished via levels indicated by l_i. An omitted level defaults to 0.

As an example, consider the rule in Line 6 of Listing 28:

```
1 { move(D,P,T): disk(D), peg(P) } 1 :- ngoal(T-1), T<=n.
```

This rule has a single head atom consisting of a cardinality constraint; it comprises all instances of `move(D,P,T)`, where `T` is constrained by the two body literals, and `D` and `P` vary over all instantiations of predicates `disk` and `peg`, respectively. Given 3 pegs and 4 disks as in Listing 29, this results in 12 instances of `move(D,P,T)` for each valid replacement of `T`, among which exactly one must be chosen according to the above rule.

Full details on the input language of *clingo* along with various examples can be found in

[1] In rule bodies, they are terminated by '`;`' or '`.`' (Gebser et al. 2015).
[2] More elaborate forms of aggregates are obtained by explicitly using function (e.g. `#count`) and relation symbols (e.g. `<=`) (Gebser et al. 2015).

the *Potassco User Guide* (Gebser et al. 2015); its semantics is given by Gebser et al. (2015) in terms of infinitary formulas.

3 Meta programming

Meta programming is a technique in which computer programs treat other programs as data (Wikipedia contributors 2021). Although this includes traditional compilers and interpreters, it has always played a prominent role in AI languages, such as Lisp or Prolog, since their syntactic proximity of program and data offers an easy way of self-modification. For instance in Prolog, meta programs allow for manipulating the execution of logic programs and constitute an easy way to extend programs with debugging information. Moreover, special-purpose predicates enable the conversion of data into new program parts during run-time.

Similarly, meta programming can be used in ASP to change the semantics of language constructs and/or implement new ones. Examples include reasoning about action and change (Baral and Gelfond 2000; Son et al. 2006; Dimopoulos et al. 2018), debugging (Gebser et al. 2008), preferences (Gelfond and Son 1997; Delgrande et al. 2003; Eiter et al. 2003) and optimization (Gebser et al. 2011), as well as guess-and-check programming (Eiter and Polleres 2006). The latter is of particular interest to us since we detail its implementation below with meta programming as well as through application interfaces in Section 8.

A common difficulty of such approaches is the conversion of programs into data, or in terms of ASP, the transformation of (non-ground) logic programs into sets of facts. Either a dedicated parser is built or a user is expected to write a program in terms of a prescribed fact format. Consequently, the resulting systems are mostly propositional and only offer a limited set of language constructs. This is because they cannot draw upon the infrastructure of an ASP system for parsing and grounding.

This issue is addressed in *clingo*, or more precisely its grounder *gringo*, by means of a fact-based representation of the grounded logic program. This enables sophisticated meta programming that may draw on the full-featured non-ground input language of *clingo*, a highly effective grounding procedure, and ultimately a factual representation reflecting all features of the input language. The remainder of this section provides an introduction to meta programming with *clingo*. The extension of logic programs during run-time is explained in the subsequent sections.

3.1 Reification format

The process of turning a (ground) logic program into a set of facts is also called *reification*. *Clingo*'s fact format of reified programs follows its intermediate language *aspif*, detailed in Appendix B.[3] This is no coincidence since both languages must capture ground programs in their full generality. In what follows, however, we concentrate on dealing with the actual logic program part and disregard non-logical statements except for `#show`.

[3] Reification was originally introduced in *clingo* 4. However, the corresponding format is different since it is modeled after the intermediate format of *smodels* (Syrjänen 2001). For details, we refer to the paper by Gebser et al. (2011).

```
1 {a}.
2 b :- a.
3 c :- not a.
```

Listing 1. A simple logic program (`ezy.lp`)

```
clingo --output=reify ezy.lp
```

Listing 2. System call to reify the logic program in Listing 1

```
1  rule(choice(0),normal(0)).
2  atom_tuple(0).     literal_tuple(0).
3  atom_tuple(0,1).

5  rule(disjunction(1),normal(1)).
6  atom_tuple(1).     literal_tuple(1).
7  atom_tuple(1,2).   literal_tuple(1,-1).

9  rule(disjunction(2),normal(2)).
10 atom_tuple(2).     literal_tuple(2).
11 atom_tuple(2,3).   literal_tuple(2,1).

13 output(a,2).

15 output(b,3).       literal_tuple(3).
16                    literal_tuple(3,3).

18 output(c,4).       literal_tuple(4).
19                    literal_tuple(4,2).
```

Listing 3. The result of the system call in Listing 2 (`ezy.rlp`)

A logic program consists of a set of rules, each of which is composed of a head and a body. While heads are formed from atoms, bodies are made of literals.

The fact format is the result of serializing the syntax tree of the ground logic program rule by rule. To this end, heads and bodies are identified via non-negative integers. Also, positive and negative integers are used to represent positive or negative literals, respectively. Hence, `0` is not a valid literal.

A rule is represented as a binary fact, using predicate `rule/2`, whose arguments reflect the head and the body of the rule. Following the rule format of *aspif*, a head is either a disjunction or a choice, which is indicated by the unary function symbols `disjunction/1` and `choice/1`. Similarly, a body is either a collection of literals or a weight constraint, indicated by functions `normal/1` and `sum/1`, respectively. All four constituents are treated as tuples, the two former consisting of atoms and the latter either of regular or weighted literals, respectively.

Let us illustrate this with the example program `ezy.lp` in Listing 1. Its (reformatted) reified program given in Listing 3 is obtained by the command in Listing 2. More precisely, the first rule in program `ezy.lp` is represented by the facts in Lines 1 to 3 of Listing 3. The fact '`rule(choice(0),normal(0)).`' states that the first rule in Listing 1 has a choice atom in the head (indicated by `choice(0)`) and a normal body (denoted by `normal(0)`). The atoms associated with a head are grouped in tuples, which are identified

by a non-negative integer. This is analogous to the treatment of literals in the body. In our example, the choice in the head is linked via the identifier `0` to a tuple of atoms declared by the fact '`atom_tuple(0).`' in Line 2 of Listing 3. The members of such an atom tuple are represented by all instances of predicate `atom_tuple/2` and share the same tuple identifier as their first argument. The atom tuple `0` has one member, as indicated by the single instance `atom_tuple(0,1)` in Line 3, where `0` stands for the tuple and `1` is the integer identifying atom `a`. To summarize, the choice atom concerns only one regular atom and this atom is identified by `1` (thus connecting `a` and `1`; see below).

Analogously, the (empty) body of the choice rule is represented by the tuple of literals that is also identified by `0`. This tuple happens to be empty, as reflected by the lack of corresponding instances of `literal_tuple/2`. Note that this tuple-centered representation treats atom and literal tuples independently and numbers them consecutively. Heads and bodies of the same rule may thus be represented by atom and literal tuples having distinct identifiers. Rules themselves have no identifier.

The second rule in Listing 1 is represented by the facts in Lines 9 to 11 of Listing 3. Unlike above, its head is a single atom and is thus represented as a one-element `disjunction` associated with atom tuple `2`. This tuple has a single element, which is accounted for by the fact `atom_tuple(2,3).` Hence, `b` is represented by `3`. Similarly, its body, also marked with `2`, comprises a single literal captured by `literal_tuple(2,1).` As above, `1` stands for the positive body literal `a`. The last rule in Listing 1 is represented analogously in Lines 5 to 7, just that its negative body literal is mapped to a negative integer, namely, `not a` is associated with `-1`.

The remaining facts in Lines 13 to 19 account for implicit output statements. This mimics the default behavior of *clingo*, outputting all atoms unless a restrictive `#show` statement is given. That is, unless any restrictions are formulated, all satisfied atoms can potentially be output. This is done by means of the binary predicate `output/2`. For instance, the output of atom `c` is linked via literal tuple `4` to integer `2` (cf. Lines 18 and 19). The indirection via the tuple representation is due to the fact that *clingo*'s output statements may be conditioned by several literals (cf. Appendix B). Finally, it is interesting to observe that no new literal tuple is generated for `output(a,2)`. Rather the one in Lines 10 and 11 is reused. This redundancy-free representation is a general feature of *clingo*'s reification.

Remark 1
Although we do not detail this here, it is worth mentioning that the reified output format of *clingo* accounts for the full spectrum of language constructs supported by *clingo*'s input language (including its generic grammar-based theory language).

In addition, *clingo* offers the options `--reify-sccs` and `--reify-steps` to calculate the strongly connected components of the ground logic program's (positive) dependency graph and to add step numbers to the reified output, in case multi-shot solving is used, respectively.

3.2 Meta encoding

The facts obtained from reifying a logic program can now be used to instantiate meta encodings. Such encodings allow us to reestablish the original or attribute a different meaning to program constructs.

```
1  conjunction(B) :- literal_tuple(B),
2        hold(L): literal_tuple(B, L), L>0;
3    not hold(L): literal_tuple(B,-L), L>0.

5  body(normal(B)) :- rule(_,normal(B)), conjunction(B).
6  body(sum(B,G))  :- rule(_,sum(B,G)),
7    #sum {
8      W,L:     hold(L), weighted_literal_tuple(B, L,W), L>0;
9      W,L: not hold(L), weighted_literal_tuple(B,-L,W), L>0
10   } >= G.

12   hold(A): atom_tuple(H,A)   :- rule(disjunction(H),B), body(B).
13 { hold(A): atom_tuple(H,A) } :- rule(     choice(H),B), body(B).

15 #show.
16 #show T: output(T,B), conjunction(B).
```

Listing 4. A simple meta program interpreting reified logic programs (`meta.lp`)

```
clingo --output=reify ezy.lp | clingo - meta.lp 0
```

Listing 5. Two steps system call using reification

To illustrate this, let us start with the simple meta encoding in Listing 4, which supports all above mentioned language constructs according to their original meaning. This encoding is only a subset; the full encoding also accounts for optimization statements (Potassco Team 2021g).

Before detailing how the encoding works, let us describe its usage. To compute the stable models of the logic program `ezy.lp` in Listing 1 via meta programming, we proceed in two steps.[4] At first, program `ezy.lp` is reified as described above, and then the resulting set of facts along with the meta encoding `meta.lp` are passed to *clingo*. The corresponding system call is given in Listing 5. The possibility of modifying the semantics of language constructs in `meta.lp` is paid by twice as much grounding effort. Interestingly, keeping the semantics as done in Listing 4 results in roughly the same solver constraints, no matter whether meta-programming is used or not. Hence, the overall overhead of meta programming is often negligible.

Remark 2
We use pipes to avoid auxiliary files. An alternative to the command in Listing 5 is

```
clingo --output=reify ezy.lp > ezy.rlp
clingo ezy.rlp meta.lp 0
```

in which the auxiliary file `ezy.rlp` is used to capture the facts in Listing 3. Note how the use of '`-`' in the pipe captures the output of the command before '`|`'.

Let us now turn to the actual meta encoding. The logic program in Listing 1 uses the unary predicate `hold/1` to express that an atom is true. Such atoms are derived in Lines 12 and 13, provided there is an original choice or disjunctive rule, whose body is

[4] Adding option `-Wno-atom-undefined` to the second call suppresses warnings due to missing definitions. The same effect is obtained by using `#defined` declarations for selected predicates.

satisfied. Both rules use conditional literals to gather all `hold` atoms belonging to the same atom tuple `H`, identifying the head of the original rule. In Line 12, this results in a disjunction of atoms, while in Line 13 all such atoms form a set of choosable atoms. In both cases, several, one, or no `hold` atoms may manifest themselves, depending on the size of the atom tuple. The satisfaction of the body of the original rule, identified by `B`, is indicated in both lines by the positive body literal `body(B)`. The corresponding atoms are derived by the two rules in Lines 5 to 10.

In Line 5, a `normal` body, composed of regular literals, is satisfied whenever all its literals are found to be true. This is realized by the rule in Lines 1 to 3 by using again conditional literals to gather all instances of `hold` atoms induced by a tuple of literals. Depending on whether the integer representing a literal is positive or negative, the corresponding `hold` atoms must be satisfied or must not be satisfied. Similarly, the rule in Lines 6 to 10 implements a weight constraint, just that the `hold` atoms are collected within a sum constraint along with their associated weights. This information is extracted from instances of the ternary predicate `weighted_literal_tuple/3` just as with `literal_tuple/2`.

Note that conjunctions of literal tuples may stand not only for rule bodies but also occur in other constructs like the conditional output statement in Line 16. Hence, it makes sense to account for them separately in Lines 1 to 3.

3.3 Examples

The next subsections give meta encodings computing classical, supported, here-and-there, and diverse models. Moreover, we show how guess-and-check programming can be addressed with both meta and application programming in Sections 3.4 and 8, respectively.

3.3.1 Classical and supported models

Let us start with some simple modifications to our meta encoding in Listing 4 that change the semantics of logic programs.

For illustration, we consider classical and supported models of logic programs. Take the logic program consisting of the following three rules:

```
a :- not b.    b :- c.    c :- b.
```

It has one stable model, {a}, two supported models, {a} and {b, c}, and three classical models, {a}, {b, c} and {a, b, c}.

This example already illustrates a general relationship between all three semantics: a stable model is also a supported model, which in turn is also a classical model but not vice versa. Intuitively, this difference is the result of how tight each semantics relates the truth of an atom to its derivability (through rules and positive body literals). While no such relation is imposed in the classical setting, supported models require that each of their atoms is supported by a rule having the atom as head and a body satisfied by the model at hand. Stable models reinforce this by stipulating that furthermore all positive body literals of the supporting rule have themselves a supporting rule and that this ends in facts (and thus yields a finite derivation). In our example, only `a` warrants this within the only stable model, while `b` and `c` only satisfy the supportedness criterion in {b, c} but lack a finite derivation. The detachment of truth from derivations (via rules) is exemplified by `a` in the classical model {a, b, c}.

```
1  #include "meta.lp".

3  atom( A ) :- atom_tuple(_,A).
4  atom(|L|) :-          literal_tuple(_,L).
5  atom(|L|) :- weighted_literal_tuple(_,L).

7  { hold(A) } :- atom(A).
```

Listing 6. Meta encoding computing classical models of logic programs (`classic.lp`)

```
1  conjunction(B) :- literal_tuple(B),
2    not not hold(L): literal_tuple(B, L), L>0;
3        not hold(L): literal_tuple(B,-L), L>0.

5  body(normal(B)) :- rule(_,normal(B)), conjunction(B).
6  body(sum(B,G))  :- rule(_,sum(B,G)),
7    #sum {
8      W,L: not not hold(L), weighted_literal_tuple(B, L,W), L>0;
9      W,L:     not hold(L), weighted_literal_tuple(B,-L,W), L>0
10   } >= G.

12   hold(A): atom_tuple(H,A)   :- rule(disjunction(H),B), body(B).
13 { hold(A): atom_tuple(H,A) } :- rule(     choice(H),B), body(B).

15 #show.
16 #show T: output(T,B), conjunction(B).
```

Listing 7. Meta encoding computing supported models of logic programs (`supported.lp`)

For computing classical models in ASP, we have to lift the ban of derivability from atoms. To this end, we extend in Listing 6 our previous meta encoding (via Line 1) with the choice rule in Line 7; its subjects are gathered in Lines 3 to 5 (by extracting the atom underlying a literal `L` by taking its absolute value `|L|`). This choice rule exempts atoms of predicate `hold/1` from having a derivation by allowing for their inclusion into a stable model at will. Classical models of a logic program can then be computed as in Listing 5, just by replacing `meta.lp` with `classic.lp`, given in Listing 6. Clearly, more direct meta encodings can be devised, for instance, by turning rules into integrity constraints.

For computing supported models, we have to make sure that each included `hold` atom is supported by some rule. The body of this rule must be satisfied, and its positive literals must themselves have supporting rules, but they do not necessarily have to yield a finite derivation. This can be accomplished by replacing the positive occurrences of `hold` literals in Lines 2 and 8 in Listing 4 by their double negation. In fact, in ASP, each true positive literal must have a non-cyclic derivation, while its double negated variant is freed from this requirement. The resulting meta encoding is given in Listing 7; cf. Lines 2 and 8 in both encodings. As above, supported models are computed by replacing `meta.lp` by `supported.lp` in the system call in Listing 5.

Remark 3
Note that the methods for computing classical and supported models may fall short in practice since reification is subject to stable-model preserving simplifications that are usually too strong for such weaker semantics. In simple cases, like ours, this can be

counterbalanced by declaring atoms as being externally defined. For instance, adding '`#external b.`' (cf. Section 5.1) to our example program spares `b` from simplification and produces the above results; otherwise not all models are obtained.

Unfortunately, such techniques become infeasible with programs using integers or function symbols since they may possess infinitely many models in general. For instance, the program consisting of '`q(f(a)).`' and '`p(X) :- p(X).`' has a single stable but infinitely many supported and classical models.

Also, note that grounding may introduce auxiliary atoms that can lead to duplicate models. To counterbalance this, one could restrict the choice in Line 7 in Listing 6 to output atoms.

3.3.2 Diverse models

Our next example application of meta programming is about computing several diverse stable models of a logic program. General approaches to computing diverse stable models can be found in the literature (Eiter et al. 2013; Romero et al. 2016).

To do so within ASP rather than by external scripting, we consider several reified stable models within one. To this end, we turn the predicate `hold` into a binary predicate, whose second argument identifies the respective stable model. These identifiers are generated in Line 1 of Listing 8, where the parameter `m` limits the number of reified stable models. The following Lines 3 to 19 constitute an extension of the original meta encoding obtained by adding an additional argument to all non-structural predicates for identifying the associated stable model. This is done throughout with variable `M`, sometimes bound by `model(M)`. Taking a logic program with n stable models and setting `m` to 2 makes Lines 1 to 19 produce n^2 stable models, each of which comprises two stable models of the original program. To distinguish the comprised models, Line 19 outputs each atom with its associated model identifier.

This initial part of Listing 8 acts as a generator of combinations of `m` stable models of the original program. In this spirit, the remainder selects two types of model combinations depending upon the setting of parameter `option` (and `k` in the first case). More precisely, the selected `m` reified stable models are

- `k`-diverse, if `option=1`, that is, the Hamming distance between each pair of the `m` stable models is greater or equal than `k`, and
- most-diverse, if `option=2`, that is, the `m` stable models maximize the sum of the Hamming distances between each pair of stable models.

Moreover, the implementation considers only atoms declared to be shown (by using predicate `show/2` rather than `hold/2` in Lines 21 to 27).

The selection of model collections having a pairwise Hamming distance greater or equal than `k` is represented by the sum constraint in Lines 22 and 25. The condition is embedded into an integrity constraint ruling out all pairs of models `M` and `N` that differ on less than `k` shown atoms. Similarly, the optimization statement in Line 27 maximizes the difference between two models; it attributes one point per difference. Given that such a statement exists for each pair of models the overall sum of differences is maximized. Note that in each case the value of parameter `option` is tested, so that at most one of them applies.

As an example, consider the logic program in Listing 9. The goal in this example is

```
model(1..m).

conjunction(B,M) :- model(M), literal_tuple(B),
      hold(L,M) : literal_tuple(B, L), L>0;
  not hold(L,M) : literal_tuple(B,-L), L>0.

body(normal(B),M) :- rule(_,normal(B)), conjunction(B,M).
body(sum(B,G),M)  :- model(M), rule(_,sum(B,G)),
  #sum {
    W,L:     hold(L,M), weighted_literal_tuple(B, L,W), L>0;
    W,L: not hold(L,M), weighted_literal_tuple(B,-L,W), L>0
  } >= G.

  hold(A,M): atom_tuple(H,A)   :- rule(disjunction(H),B), body(B,M).
{ hold(A,M): atom_tuple(H,A) } :- rule(     choice(H),B), body(B,M).

show(T,M) :- output(T,B), conjunction(B,M).
#show.
#show (T,M): show(T,M).

:- model(M), model(N), M<N, option=1,
   #sum {
     1,T:     show(T,M), not show(T,N);
     1,T: not show(T,M),     show(T,N)
   } < k.

#maximize { 1,T,M,N: show(T,M), not show(T,N), model(N), option=2 }.
```

Listing 8. Meta encoding computing several (diverse) stable models (`many.lp`)

```
{ x((1..n,1..n)) } = c.

connect(C) :- x(C), C<=C': x(C').
connect((X',Y')) :- connect((X,Y)), x((X',Y')), |X-X'|+|Y-Y'|=1.

:- x(C), not connect(C).

#show x/1.
```

Listing 9. Mark `c` cells of an `n`×`n` grid that must be connected to each other (`cells.lp`)

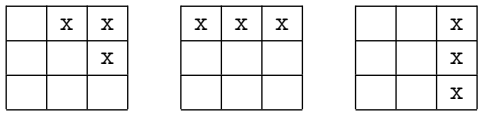
to mark `c` cells of an `n`×`n` grid such that the marked cells are connected to each other (where `c` and `n` are parameters).

Let us begin by computing three different stable models with a pairwise Hamming distance greater or equal than one:

```
UNIX> clingo --output=reify cells.lp -c n=3 -c c=3 | \
      clingo - many.lp -c option=1 -c k=1 -c m=3
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
(x((2,3)),1) (x((3,2)),1) (x((3,3)),1) \
(x((1,3)),2) (x((2,3)),2) (x((3,3)),2) \
```

```
(x((3,1)),3) (x((3,2)),3) (x((3,3)),3)
SATISFIABLE
```

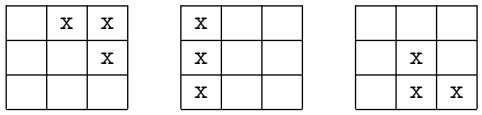
The obtained three reified stable models can be visualized as follows (by letting (1,1) be the lower left corner):



Next, consider the result obtained by imposing a Hamming distance of `6`:

```
UNIX> clingo --output=reify cells.lp -c n=3 -c c=3 | \
      clingo - many.lp -c option=1 -c m=3 -c k=6
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
(x((2,3)),1) (x((3,2)),1) (x((3,3)),1) \
(x((1,1)),2) (x((1,2)),2) (x((1,3)),2) \
(x((2,1)),3) (x((2,2)),3) (x((3,1)),3)
SATISFIABLE
```

Unlike above, the three solutions do not overlap and possess the imposed pairwise Hamming distance:

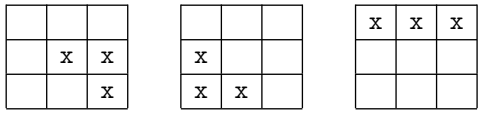


Note that no three stable models are obtainable with a pairwise Hamming distance exceeding 6.

A similar result is obtained by maximizing the sum of Hamming distances, as shown next. We use `--quiet=1,2,2` to suppress intermediate models:

```
UNIX> clingo --output=reify cells.lp -c n=3 -c c=3       | \
      clingo - many.lp -c option=2 -c m=3 --quiet=1,2,2
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 5
(x((2,2)),1) (x((3,1)),1) (x((3,2)),1) \
(x((1,1)),2) (x((1,2)),2) (x((2,1)),2) \
(x((1,3)),3) (x((2,3)),3) (x((3,3)),3)
OPTIMUM FOUND
```

The obtained solution has the same quality as the previous one, namely, `18`, three times a Hamming distance of six. Note that this property is a particularity of our example:



3.3.3 Here-and-there models

The logical foundations of ASP rest upon the logic of Here-and-There (HT; Heyting 1930), or more specifically its non-monotonic extension called Equilibrium Logic (Pearce 1997). Informally, interpretations in HT consist of pairs of interpretations (H, T) such

```
1  atom( A ) :- atom_tuple(_,A).
2  atom(|L|) :-          literal_tuple(_,L).
3  atom(|L|) :- weighted_literal_tuple(_,L).

5  model(h).  model(t).

7  { hold(A,h) } :- atom(A),    option=1.
8  { hold(A,t) } :- atom(A).
9  :- hold(L,h), not hold(L,t).

11 :- not hold(L,h), hold(L,t), option=3.

13 conjunction(B,M) :- model(M), literal_tuple(B),
14       hold(L,M): literal_tuple(B, L), L>0;
15   not hold(L,t): literal_tuple(B,-L), L>0.

17 body(normal(B),M) :- rule(_,normal(B)), conjunction(B,M).
18 body(sum(B,G),M)  :- model(M), rule(_,sum(B,G)),
19   #sum {
20     W,L:     hold(L,M), weighted_literal_tuple(B, L,W), L>0;
21     W,L: not hold(L,t), weighted_literal_tuple(B,-L,W), L>0
22   } >= G.

24 hold(A,M): atom_tuple(H,A) :- rule(disjunction(H),B), body(B,M).
25 hold(A,M); not hold(A,t)   :- atom_tuple(H,A),
26                               rule(     choice(H),B), body(B,M).

28 #show.
29 #show (T,M): output(T,B), conjunction(B,M).
```

Listing 10. A meta encoding for computing here-and-there models (`ht.lp`)

that $H \subseteq T$. The intuition of using two such sets is that atoms in H are the ones that can be proved, atoms not in T are those for which no proof exists, and, finally, atoms in $T \setminus H$ are assumed to hold but have not been proved. A total HT model (T, T) is an equilibrium model if there is no HT model (H, T) with $H \subset T$. In such a case, T is also called a stable model. A comprehensive account of HT is given by Pearce (1997).

The meta encoding for computing HT models of logic programs is given in Listing 10; it builds upon several constructions already used in the previous meta encodings. The first part in Lines 1 to 8 is similar to the computation of classical models in Listing 6 in generating all admissible HT interpretations. As above, we use terms, namely, `h` and `t`, to distinguish the components in HT interpretations such as (H, T). Line 7 generates atoms in H, Line 8 the ones in T, and the integrity constraint in Line 9 makes sure that $H \subseteq T$. The type of generated HT interpretations is once more determined by parameter `option`. A generated pair of interpretations (H, T) is

- an HT interpretation, if `option=1`,
- an HT interpretation with minimal $H \subseteq T$, if `option=2` (or undefined), or
- an equilibrium model, if `option=3`.

The aforementioned description captures the first case. In both remaining cases, the free

generation of atoms in H is dropped (cf. Line 7) and they must rather be derived via program rules. In addition, option value `3` enforces $T \subseteq H$ via the integrity constraint in Line 11.

The satisfaction of rules and the derivation of `hold` atoms in Lines 13 to 26 is similar to the computation of diverse models in Listing 8 with a few exceptions due to the different semantic setting. First of all, a negative literal only holds in an HT interpretation (H, T) if its atom does not belong to T. Accordingly, the tests for negative body literals in Lines 15 and 21 only refer to `hold` atoms associated with `t`. Second, the choice of an atom a amounts in HT to an instance of the law of the excluded middle $a \vee \neg a$. This classical treatment of an atom leaves only two possibilities in Line 25, either a is false, and thus does not belong to T (and neither to H), or a is true in H or T.[5] Note that whenever `option=1`, this variant of our meta encoding merely checks the satisfaction of rules in both components of an HT interpretation. Unlike this, `hold` atoms associated with `h` must be derived via the meta encoding in all other cases.

For illustration, consider the logic programs `or.lp` containing the disjunction '`a;b.`' and `even.lp` with rules '`a :- not b. b :- not a.`'. Both programs are equivalent in the sense that they share the same stable models {`a`} and {`b`}. However, putting each program together with rules '`a :- b. b :- a.`' yields different stable models, indicating that both programs are not interchangeable in an encompassing program; in formal terms, they are not strongly equivalent (Lifschitz et al. 2001). Interestingly, the strong equivalence of logic programs corresponds to their equivalence in HT (their mere equivalence can be read off the same set of equilibrium models).

Let us verify this by means of our meta encoding in Listing 10.

Program `or.lp` has five HT models:

```
UNIX> clingo --output=reify or.lp | \
      clingo - ht.lp 0 -c option=1
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
(b,t) (b,h)
Answer: 2
(b,t) (a,t) (b,h)
Answer: 3
(a,t) (a,h)
Answer: 4
(b,t) (a,t) (a,h)
Answer: 5
(b,t) (a,t) (b,h) (a,h)
SATISFIABLE
```

In contrast to '`a;b.`', program `even.lp` has the additional HT model $(\emptyset, \{\mathtt{a}, \mathtt{b}\})$, as witnessed by the first of its six HT models:

```
UNIX> clingo --output=reify even.lp | \
      clingo - ht.lp 0 -c option=1
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
(a,t) (b,t)
```

[5] Strictly speaking, the rule in Lines 25 and 26 could be specialized by adding `M=h`, since the instance with `M=t` is subsumed by the rule in Line 8, which is equivalent to '`hold(A,t); not hold(A,t) :- atom(A).`'.

```
Answer: 2
(a,t) (a,h)
Answer: 3
(a,t) (b,t) (a,h)
Answer: 4
(b,t) (b,h)
Answer: 5
(a,t) (b,t) (b,h)
Answer: 6
(a,t) (b,t) (a,h) (b,h)
SATISFIABLE
```

Unlike above, both programs have the same equilibrium models, which establishes their equivalence. This can be verified using option value `3` instead of `1` above.

Further examples of meta programming are available online (Potassco Team 2021g).

3.4 Guess-and-check programming

So far, we addressed problems at the first level of the polynomial hierarchy, sharing the same complexity as normal logic programs in ASP (Dantsin et al. 2001). In fact, ASP can also be used for expressing problems at the second level, when disjunctive heads or non-monotonic aggregates are used.

An interesting class of such problems consists of two subproblems (Eiter and Polleres 2006): A guess-and-check logic program is a pair (P, Q) of normal logic programs whose solution is a stable model of P that results in an unsatisfiable program once its atoms are added as facts to Q. This combines a satisfiability problem with an unsatisfiability problem, in the most interesting case, an *NP* with a *coNP* problem since this lifts the joined problem to the second level of the polynomial hierarchy. An example of this is preference handling, where the first problem defines feasible solutions, while the second one ensures that there are no better solutions. Another example is (bounded) conformant planning under incomplete information, where the first problem gives a plan in some scenario, while the second one makes sure that it is not invalidated in any other scenario. We are interested in finding a single ASP encoding using disjunctive heads that combines both problems and yields solutions despite the unsatisfiability of the second subproblem.

For implementing such problems, Eiter and Gottlob (1995) invented the *saturation* technique, using the elevated complexity of disjunctive logic programming. In stark contrast to the ease of common ASP modeling, however, this technique is rather involved and hardly usable by ASP laymen. This shortcoming was addressed by means of meta programming by Eiter and Polleres (2006). The idea is to represent both components of a guess-and-check program as facts and to combine them with a meta encoding to obtain a single joint program after grounding. The meta encoding implements the saturation technique and discharges users from this intricate modeling task. Now, the reification functionalities of *clingo* allow us to simplify this even further by relieving users from the specification of guess-and-check program in terms of facts.

In what follows, we showcase the computation of solutions to guess-and-check programs by means of meta programming with *clingo*. To this end, let us start with a sketch of the meta encoding implementing saturation and tie it up to guess-and-check programming afterwards. We build on an encoding of saturation put forward by Gebser et al. (2011)

```
1 { a(1..2) }.
```

Listing 11. Guess program (`guess.lp`)

```
:- not a(1).
```

Listing 12. Check program (`check.lp`)

```
{ a(1..2) }.
```

Listing 13. Import guess atoms (`in.lp`)

```
:- not bot.
```

Listing 14. Enforce `bot` atom (`bot.lp`)

```
bot :- output(a(X),B),     a(X), fail(normal(B)).
bot :- output(a(X),B), not a(X), true(normal(B)).
```

Listing 15. Synchronize stable models (`glue.lp`)

and present in Listing 47 in Appendix A a revised version based on the *aspif* format.[6] A detailed account of the saturation technique goes beyond the scope of this tutorial (see the papers by Eiter and Gottlob (1995) and Eiter and Polleres (2006) for details). From the perspective of plain ASP, a saturation-based meta encoding acts as an ordinary one just that it yields the set of all atoms whenever a program is unsatisfiable. That is, it generates one stable model for each original stable model, whenever the program is satisfiable, and otherwise, it yields a unique stable model containing the set of all atoms of the program. Also, unlike Listing 4, the encoding in Listing 47 takes a reified normal logic program as input and results in a disjunctive logic program after grounding.

Now, for implementing guess-and-check programming, we exploit the property of saturation-based meta encodings that the non-existence of stable models results in a set containing all atoms. In such a case, the encoding in Listing 47 additionally produces the special-purpose atom `bot` for indicating unsatisfiability; this atom never appears in a genuine stable model. Hence, to make sure that an (augmented) check program is unsatisfiable, we can simply add an integrity constraint that enforces `bot` to hold (cf. Listing 14).

What remains to be done is to account for the import of the guessed atoms into the check program and to align the stable models of the guess and the check program. For simplicity, we address both tasks in a rather direct way and show below a more principled alternative. For addressing the first task, we simply add a choice rule over the possible guess atoms to the check program to allow for exchanging all possible interpretations. For this to work, however, the guess atoms must not occur among the heads of the check program. Otherwise, a guess atom may be false but become true in the check program. For illustration, consider the simple guess and check programs in Listings 11 and 12 along with the import of guess atoms into the check program in Listing 13.

The second task, synchronizing the respective stable models, is handled by saturation. Remember that the check program uses a fresh set of guess atoms derived via the choice rule in Listing 13. Hence, we have two different sets of atoms in both the guess and the check programs, which have to be synchronized. In valid counter examples, the truth values of both sets of atoms have to agree. We simply derive the `bot` in Listing 15 if the atoms differ, which triggers saturation and discards invalid candidates for counter examples.

[6] The original meta encoding (Gebser et al. 2011) relies on the *smodels* format used for reification in *clingo* 4 (cf. Footnote 3).

The body literals of the rules in Listing 15 already hint at the design decision to only reify the check program, while leaving the guess program intact. At last, let us put all this together for our example guess and check program, where program `metaD.lp` is given in Listing 47:

```
UNIX> clingo --output=reify --reify-sccs check.lp in.lp | \
      clingo - metaD.lp bot.lp glue.lp guess.lp 0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
a(2)
SATISFIABLE
```

Combining program `guess.lp` and `check.lp` from Listings 11 and 12 in a guess-and-check program eliminates all stable models of `guess.lp` that contain `a(1)`. Note that joining both in a regular program eliminates models excluding `a(1)`.

The first call to *clingo* merely reifies program `check.lp`, so that it gets interpreted by the meta encoding `metaD.lp` in the second call. Unlike this, program `guess.lp` remains untouched. Hence, the problem handed to the second call of *clingo* combines both the guess and the check program, whereby the latter is encoded via saturation in order to succeed upon unsatisfiability. And finally, program `glue.lp` is in charge of aligning the guessed stable models to the checker by saturating the non-aligned ones. Clearly, our illustrative example is very simple and could also be solved directly. In what follows, we present some more substantial use-cases.

Remark 4
It is instructive to observe the effect of programs `bot.lp` and `glue.lp` on the formation of the joint solving result:

- Without both `bot.lp` and `glue.lp`, we get all combinations of stable models of the guess program with stable models of the check program (choice rules included), if the check program is satisfiable, and with the unique saturated set of atoms containing `bot`, otherwise.
- Without `bot.lp` but with `glue.lp`, we get all combinations of stable models of the guess program with stable models of the check program (conjoined with all facts stemming from guess atoms), if this program is satisfiable, and with the unique saturated set of atoms containing `bot`, otherwise.
- With both `bot.lp` and `glue.lp`, we get all stable models of the previous item that contain `bot`. That is, we get all combinations of stable models of the guess program with the unique saturated set of atoms containing `bot` whenever the check program (conjoined with all facts stemming from guess atoms) is unsatisfiable, and no stable models, otherwise.

A more generic approach can be obtained by capturing the guessed atoms by a dedicated predicate, say `guess/1`. This also eases the restriction of the relayed atoms to a distinguished subset.

In our example, the guess program in Listing 11 is then augmented with the specification of the actual guess in Listing 16. Accordingly, the guessed atoms become enveloped in predicate `guess/1` in the check program from Listing 12, viz. '`:- not guess(a(1)).`'. Similarly, the import of guessed atoms is modified in the program in Listing 17. This

```
guess(a(X)) :- a(X).
```

Listing 16. Export guess atoms (`out.lp`)

```
{ guess(X) } :- output(guess(X),_).
```

Listing 17. Import guess atoms (`in.lp`)

```
bot :- output(guess(X),B),     guess(X), fail(normal(B)).
bot :- output(guess(X),B), not guess(X), true(normal(B)).
```

Listing 18. Generic synchronization of stable models (`superglue.lp`)

version of program `in.lp` replaces the specific choice rule '`{ a(1..2) }.`' in Listing 13 by a generic rule. The instantiation of the first rule is obtained from the symbol table of the guess program, as shown below. With this approach, both the guess as well as the check programs come in pairs, the former accompanied by the specification of guessed atoms in `out.lp` in Listing 16, and the latter with choices delineating all possible sets of guessed atoms in `in.lp` in Listing 17. While the first fixes all guessable atoms, the second one is generic. And finally, also the synchronization of stable models can now be expressed in a generic way via the rules in Listing 18. Note that except for Listing 16, all auxiliary programs are problem independent.

Putting everything together, we can solve the example in Listings 11 to 12:

```
UNIX> clingo --output=reify                                          \
             guess.lp out.lp <(echo "#show guess/1.")              | \
      grep "output(guess(.*))"                                     | \
      clingo --output=reify --reify-sccs                             \
             - check.lp in.lp                                      | \
      clingo - metaD.lp bot.lp superglue.lp guess.lp out.lp 0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
a(2)
SATISFIABLE
```

The purpose of the first two lines in the system call above is to extract all guessable atoms from the symbol table of the guess program. (In passing, this illustrates a pragmatic way of exploiting the symbol table of reified programs.) In our example, this results in the facts `output(guess(a(1)),5).` and `output(guess(a(2)),6).` These atoms are then used to instantiate the rules in Listing 17 and 18. Otherwise, the two remaining calls work just as described above, and obviously yield the same result.

In fact, the effort of encapsulating the guessed atoms along with the resulting generality pays off, as we demonstrate in the following two use-cases. More examples of guess-and-check programming are available online (Potassco Team 2021g).

Preferences. In this example, we use the above setup to compute subset maximal stable models of logic programs. The idea is to guess a candidate stable model and to check whether any of its proper supersets is a stable model of the program, too. If this fails, the candidate is subset maximal.

To illustrate this, reconsider program `guess.lp` in Listing 11 along with the specification of guessable atoms in Listing 16. The check program consists once more of program `guess.lp` yet extended with the rules in Listing 19.

The atom `better` is derived whenever the stable model generated by the check program

```
1 better :- a(Y), not guess(a(Y)), a(X): guess(a(X)).
2 :- not better.
```

Listing 19. Identifying subset maximal stable models (`superset.lp`)

```
1 #const n=3.
2 n { o(1..n,1..n) } n.
3 :- not win.
4 win :- I=1..n, o(I,J): J=1..n.
5 win :- J=1..n, o(I,J): I=1..n.
6 win :- o(I,I): I=1..n.
7 win :- o(I,n+1-I): I=1..n.

9 #show o/2.
```

Listing 20. Player O (`playero.lp`)

```
1 #const n=3.
2 n { x(1..n,1..n) } n.
3 :- not win.
4 win :- I=1..n, x(I,J): J=1..n.
5 win :- J=1..n, x(I,J): I=1..n.
6 win :- x(I,I): I=1..n.
7 win :- x(I,n+1-I): I=1..n.

9 :- guess(o(I,J)), x(I,J).
```

Listing 21. Player X (`playerx.lp`)

```
1 guess(o(I,J)) :- o(I,J).
```

Listing 22. Exporting guess atoms (`out.lp`)

is a proper superset of the guessed stable model. This candidate model is not subset maximal if some strictly larger model is obtainable, as checked in Line 2 of Listing 19.

What makes this example different from the one above (and below) is that the guess and the check programs are based on the same logic program and thus deal with the same set of stable models. A guessed stable model is optimal, if no better model can be obtained by the checker. This nicely reflects the checker's role of generating potential counterexamples.

Proceeding in the same manner as above, we compute the only subset maximal model of program `guess.lp` as follows:

```
UNIX> clingo --output=reify guess.lp out.lp                    \
             <(echo "#show guess/1.")                          | \
      grep "output(guess(.*))"                                 | \
      clingo --output=reify --reify-sccs                         \
             - guess.lp superset.lp in.lp                      | \
      clingo - metaD.lp bot.lp superglue.lp guess.lp out.lp 0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
a(1) a(2)
SATISFIABLE
```

Tic-tac-toe. In this example, we consider a simplified 3×3 Tic-tac-toe puzzle. Player O has to place their three tokens in a winning configuration such that afterwards Player X cannot place theirs in a winning position. Hence, the game is not played in turns and just guaranteed winning configurations of Player O are determined.

This example involves two similar yet different programs. Unlike above, a guess involves only a subset of the atoms of the generated stable models.

The encodings of Player O and X are given in Listings 20 and 21. The former acts as a guess program and the latter as a checker program; only the positions of Player O

are passed to Player X, as fixed in Listing 22. In this way, the positions occupied by O become blocked for Player X in Line 9 of Listing 21.

Relying on the above setup, we can then compute the two undefeatable diagonal configurations in Tic-tac-toe:

```
UNIX> clingo --output=reify playero.lp out.lp                    \
             <(echo "#show guess/1.")                          | \
      grep "output(guess(.*))"                                 | \
      clingo --output=reify --reify-sccs                         \
             - playerx.lp in.lp                                | \
      clingo - metaD.lp bot.lp superglue.lp playero.lp out.lp 0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
o(1,1) o(2,2) o(3,3)
Answer: 2
o(1,3) o(2,2) o(3,1)
SATISFIABLE
```

4 About *clingo* applications

We have seen in the last section how the functionality of ASP systems can be changed by using meta programming. In particular, the reification of logic programs allows us to control ASP by means of ASP. The remainder of this tutorial parallels this by showcasing several ways of how ASP can be managed with other programming languages. As mentioned in the introduction, we have chosen Python as our example language, although other choices are possible.

This section focuses on the overall setup. The following ones delve into particular functionalities and case studies.

Clingo offers three ways of combining ASP with other programming languages, either via an embedded script, module import, or its application class. Although all three options allow us to change the behavior of *clingo* by overwriting its main function, they aim at rather different use cases. We discuss the three choices below and show how they treat the common example in Listing 23. The idea is to outsource the computation of divisors to

```
1 num(3).
2 num(6).
3 div(N,@divisors(N)) :- num(N).
```

Listing 23. Example with external function (`example.lp`)

Python via an external function call. Such calls look like function terms but are preceded by `@` (Gebser et al. 2015). In our example, the term `@divisors(N)` in Line 3 assumes the definition of a corresponding method in Python that takes the instantiation of `N` as argument; its results are added as a term pool (Gebser et al. 2015) so that one or several values can be accommodated. In our example, the head of the third rule thus results in the atoms `div(3,(1;3))` and `div(6,(1;2;3;6))`.

4.1 Embedded Python code

The simplest way to extend an ASP encoding with a Python method is to add it as an embedded script.

In our example, this can be done by supplying *clingo* with the embedded Python script in Listing 24. It shows that foreign language scripts are enclosed in `#script` and `#end.`

```
#script (python)

from clingo.symbol import Number

def divisors(a):
    a = a.number
    for i in range(1, a+1):
        if a % i == 0:
          yield Number(i)

#end.
```

Listing 24. Embedded Python code (`embedded.lp`)

and supplied with an argument indicating the used language. The code in this block is arbitrary and executed before grounding. Functions defined in it, such as the `divisors` function, can be called from ASP by prepending an `@` symbol. Finally, such scripts are meant to be part of the input of *clingo* just as the encoding in Listing 23, as shown below:

```
UNIX> clingo example.lp embedded.lp
clingo version 5.5.0
Reading from example.lp ...
Solving...
Answer: 1
num(3) num(6) div(3,1) div(3,3) div(6,1) div(6,2) div(6,3) div(6,6)
SATISFIABLE
```

Such external term evaluation (during grounding) is an intended use case for embedded scripts. The *clingo* object invoked on the command line is in charge of loading, grounding, and solving, unless these tasks are taken from it (Gebser et al. 2019). Hence, the usage of embedded scripts is generally most suitable for small extensions to logic programs, anything on the term level during grounding. Often they are used to perform calculations that are difficult or inconvenient to express in ASP.

4.2 The clingo Python module

The second alternative is to write a Python script using *clingo*'s Python module. The module provides high level functions to interact with the grounder and solver including input and output processing as well as fine-grained control over the grounding and solving process. This provides a convenient way to use *clingo* as part of a larger project. The surrounding application is in charge of the control flow and ASP is used to perform specific computations. Even for simple computations, this avoids error prone string processing like transforming data into ASP facts or parsing the solver's output.

Listing 25 implements the previous example by using the Python module. The main difference is that we have to construct a `Control` object and take care of the control flow

```
1  from clingo.symbol import Number
2  from clingo.control import Control

4  class ExampleApp:
5      @staticmethod
6      def divisors(a):
7          a = a.number
8          for i in range(1, a+1):
9              if a % i == 0:
10               yield Number(i)

12     def run(self):
13         ctl = Control()
14         ctl.load("example.lp")
15         ctl.ground([("base", [])], context=self)
16         ctl.solve(on_model=print)

18 if __name__ == "__main__":
19     ExampleApp().run()
```

Listing 25. Example with external function (`module.py`)

ourselves. Such a `Control` object encapsulates an instance of *clingo*. First, the program is added to the `Control` object using its `load` function. Then, the `base` part is grounded.[7] To be able to call the `divisors` function, we pass the self argument as context to the ground function. Finally, note that we use Python's `print` function as `on_model` callback. An `on_model` callback is a function passed to `solve` that is called for each model. It allows for inspecting (and printing) the current model. This construction is necessary because, unlike with the *clingo* system, there is no output foreseen when using the *clingo* module in Python. This is nicely reflected by the plain output produced by Python's `print` function when solving our example with Listing 25:

```
UNIX> python module.py
num(3) num(6) div(3,1) div(3,3) div(6,1) div(6,2) div(6,3) div(6,6)
```

This is different from the ASP-specific output produced by *clingo* in the previous section. While there ASP is extended with Python, here it is the other way around. This renders the use of ASP completely opaque.

4.3 Implementing a system based on clingo

Finally, we present a third way that aims at building custom systems based on *clingo*. This is similar to embedded Python code but gives more control to customize the system. For example, parts of the text output can be modified, additional options can be registered, or the way input files are treated can be changed completely.

Unlike the previous example, we now derive our `ExampleApp` class from *clingo*'s application class. The resulting class can then be used with the `clingo_main` function, which starts a process similar to the one in *clingo* but possibly with some overwritten functions.

[7] Without any declarations, rules belong to a logic program referred to as `base` (cf. Section 5.1).

First, the program name and version are defined. These values are then used in the status, help, and version output of *clingo*. Furthermore, each application class must implement a main function, which is called right after option parsing and is in charge of the grounding and solving process. The function receives a `Control` object and a list of paths to the files passed on the command line. The subsequent code processes the files just as *clingo* would. Similar to the previous example, we pass in Listing 26 the `ExampleApp` object to the ground function to be able to call its `divisors` method during grounding.

```
1  import sys
2  from clingo.symbol import Number
3  from clingo.application import Application, clingo_main

5  class ExampleApp(Application):
6      program_name = "example"
7      version = "1.0"

9      @staticmethod
10     def divisors(a):
11         a = a.number
12         for i in range(1, a+1):
13             if a % i == 0:
14                 yield Number(i)

16     def main(self, ctl, files):
17         for path in files: ctl.load(path)
18         if not files:
19             ctl.load("-")
20         ctl.ground([("base", [])], context=self)
21         ctl.solve()

23 if __name__ == "__main__":
24     clingo_main(ExampleApp(), sys.argv[1:])
```

Listing 26. Example application (`app.py`)

Note that we do not need to use a print function to output models. Rather the one of *clingo* is used in a seamless way. Again, this is reflected by the output of solving our example with Listing 26:

```
UNIX> python app.py example.lp
example version 1.0
Reading from example.lp
Solving...
Answer: 1
num(3) num(6) div(3,1) div(3,3) div(6,1) div(6,2) div(6,3) div(6,6)
SATISFIABLE
```

Furthermore, this output already hints at the benefits of using *clingo*'s `Application`. While control is exercised from Python, as in the last section, it allows us to draw on *clingo*'s infrastructure, similar to using embedded scripting.

The utility of this class becomes apparent in the rest of the tutorial, where it is used throughout as the basic building block of all *clingo* based systems.

5 Multi-shot ASP solving

Multi-shot solving allows for solving continuously changing logic programs in an operative way. This can be controlled via APIs implementing reactive procedures that loop on grounding and solving while reacting, for instance, to outside changes or previous solving results. Such reactions may entail the addition or retraction of rules that *clingo*'s operative approach can accommodate while leaving unaffected program parts intact within the solver. This avoids re-grounding and solving benefits from heuristic scores and constraints learned over time.

We begin with an informal overview of the central features and language constructs of *clingo*'s multi-shot solving capabilities. We illustrate them in Sections 5.2 and 5.3 by showcasing two exemplary reasoning modes, namely branch-and-bound-based optimization and incremental ASP solving. A comprehensive introduction to multi-shot solving with *clingo* is given by Gebser et al. (2019).

5.1 A gentle introduction

Clingo allows us to structure (non-ground) rules into subprograms. To this end, a program can be partitioned into several subprograms by means of the directive `#program`; it comes with a name and an optional list of parameters. Once given in the input, the directive gathers all rules up to the next such directive (or the end of file) within a subprogram identified by the supplied name and parameter list. As an example, we specify two subprograms `base` and `acid(k)` in file `chemistry.lp` in Listing 27. Note that `base` is a

```
1 a(1).
2 #program acid(k).
3 b(k).
4 c(X,k) :- a(X).
5 #program base.
6 a(2).
```

Listing 27. Subprograms `base` and `acid(k)` (`chemistry.lp`)

special subprogram (with an empty parameter list). In addition to the rules in its scope, it gathers all rules not preceded by any `#program` directive. Hence, in the above example, the `base` subprogram includes the facts `a(1)` and `a(2)`, although, only the latter is in the actual scope of the directive in Line 5. Without further control instructions (see below), *clingo* grounds and solves the `base` subprogram only, essentially, yielding the standard behavior of ASP systems. The processing of other subprograms such as `acid(k)` is subject to external governance.

We first have a look at customizing grounding and solving by creating a `Control` object, as put forward in Section 4.2. For illustration, let us consider two Python code snippets:[8]

```
1 from clingo.control import Control
2 ctl = Control()
3 ctl.load("chemistry.lp")
```

[8] The `ground` routine takes a list of pairs as argument. Each such pair consists of a subprogram name (e.g. `base` or `acid`) and a list of actual parameters (e.g. `[]` or `[Number(42)]`).

```
4  ctl.ground([("base", [])])
5  ctl.solve(on_model=print)
```

While the above control program matches the default behavior of *clingo*, the one below ignores all rules in the `base` program but rather contains a `ground` instruction for `acid(k)` in Line 5, where the parameter `k` is to be instantiated with the term `42`.

```
1  from clingo.symbol import Number
2  from clingo.control import Control
3  ctl = Control()
4  ctl.load("chemistry.lp")
5  ctl.ground([("acid",[Number(42)])])
6  ctl.solve(on_model=print)
```

The treatment of parameter `k` is similar to that of a constant, defined with `#const` (Gebser et al. 2015), yet restricted to the rules in the scope of the respective subprogram. Accordingly, the schematic fact `b(k)` is turned into `b(42)`. No ground rule is obtained from '`c(X,k) :- a(X)`' due to lacking instances of `a(X)`. Hence, the `solve` call in Line 6 yields a stable model consisting of `b(42)` only. Note that `ground` instructions apply to the subprograms given as arguments, while `solve` triggers reasoning with respect to all accumulated ground rules.

For more elaborate reasoning processes, it is indispensable to activate and/or deactivate ground rules on demand. For instance, former initial or goal state conditions may need to be relaxed or completely replaced when modifying a planning problem, e.g., by extending the plan length. To expire transient rules, *clingo* provides the `#external` directive. This directive goes back to *lparse* (Syrjänen 2001), where it was used to exempt (input) atoms from simplifications during grounding. Its functionality is generalized in *clingo* to provide a flexible handling of yet undefined atoms in the course of grounding and solving.

For continuously assembling ground rules evolving at different stages of a reasoning process, `#external` directives declare atoms that may still be defined by rules added later on. In terms of module theory (Oikarinen and Janhunen 2006) such atoms correspond to inputs, which (unlike undefined output atoms) must not be simplified. For declaring such input atoms, *clingo* offers schematic `#external` directives that are instantiated along with the rules of their respective subprograms.

For instance, the directive in the second line below

```
8  #program acid(k).
9  #external d(X,k): c(X,k).
10 e(X,k) :- d(X,k).
```

is treated similar to the rule '`d(X,k) :- c(X,k)`' during grounding, just that only the head atoms of the resulting ground instances are collected as inputs. Hence, adding the above lines to program `chemistry.lp` and grounding both subprograms `base` and `acid(42)` yields the external atoms `d(1,42)` and `d(2,42)`. Thus, we furthermore obtain the ground rules '`e(1,42) :- d(1,42)`' and '`e(2,42) :- d(2,42)`'.

Once grounded, the truth value of external atoms can be changed via *clingo*'s API (until the atoms become defined by corresponding rules or are released). By default, the initial truth value of external atoms is set to false. Then, for example, with *clingo*'s Python API, the call `ctl.assign_external(d(2,42),True)`[9] can be used to set the truth value of the

[9] For constructing atoms, symbolic terms, or function terms, respectively, the *clingo* API function

external atom `d(2,42)` to true. This can be used to activate and deactivate rules in logic programs. For instance, the rule '`e(1,42) :- d(1,42)`' is ineffective because `d(1,42)` is false by default. Hence, a subsequent `solve` call yields the stable model consisting of atoms `a(1)`, `a(2)`, `c(1,42)`, `c(2,42)`, `d(2,42)`, and `e(2,42)`. One further interesting use case is to release external atoms. The call `ctl.release_external(d(1,42))` removes the external atom and the rule '`e(1,42) :- d(1,42)`' from the `Control` object.

Remark 5
Module theory (Oikarinen and Janhunen 2006) is used to characterize the composition of ground subprograms during multi-shot solving. For this, each ground subprogram is associated with a module. Accordingly, the restrictions of module composition apply: First, no two subprograms may define the same atom. Second, loops cannot spread across subprograms. We refer to the paper by Gebser et al. (2019) for details.

The first condition can be enforced by equipping rule heads with parameters, as done above. Often a natural choice for this is an argument identifying a step, as `t` in Listing 31.

5.2 Branch-and-bound-based optimization

In this section and the following one, we illustrate *clingo*'s multi-shot solving machinery by applying it to a Towers of Hanoi puzzle (Potassco Team 2021h). Our example consists of three pegs and four disks of different size; it is shown in Figure 1. The goal is to move all disks from the left peg to the right one. Only the topmost disk of a peg can be moved at a time. Furthermore, a disk cannot be moved to a peg already containing a disk of

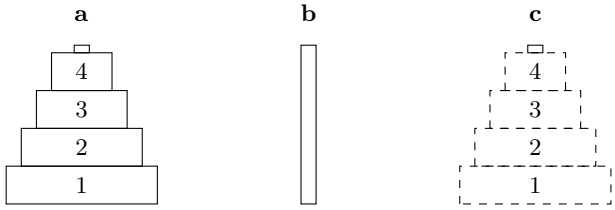


Fig. 1. Towers of Hanoi: initial and goal situation

smaller size. Although there is an efficient algorithm to solve our simple puzzle, we do not exploit it here and merely specify conditions for sequences of moves being solutions. The Towers of Hanoi puzzle constitutes a typical planning problem, aiming at finding a plan, that is, a sequence of actions, that leads from an initial state to a state satisfying a goal.

To illustrate how multi-shot solving can be used for realizing branch-and-bound-based optimization, we consider the problem of finding the shortest plan solving our puzzle within a given horizon. To this end, we adapt the Towers of Hanoi encoding by Gebser et al. (2019) in Listing 28. Here, the length of the horizon is given by parameter `n`. The problem instance in Listing 29 together with Line 1 in Listing 28 gives the initial configuration of disks in Figure 1. Similarly, the goal is checked in Lines 3–4 of Listing 28 (by drawing on the problem instance in Listing 29). Because the overall objective is to

`Function` has to be used. Similarly, numeric terms are constructed using function `Number`. Hence, the expression `d(2,42)` actually stands for `Function("d", [Number(2),Number(42)])`.

```
1  on(D,P,0) :- init_on(D,P).

3  ngoal(T) :- goal_on(D,P), T=0..n, not on(D,P,T).
4  :- ngoal(n).

6  1 { move(D,P,T): disk(D), peg(P) } 1 :- ngoal(T-1), T<=n.

8  move(D,T)        :- move(D,P,T).
9  on(D,P,T)        :- move(D,P,T).
10 on(D,P,T)        :- on(D,P,T-1), not move(D,T), T<=n.
11 blocked(D-1,P,T) :- on(D,P,T-1).
12 blocked(D-1,P,T) :- blocked(D,P,T), disk(D).

14 :- move(D,P,T), blocked(D-1,P,T).
15 :- move(D,T), on(D,P,T-1), blocked(D,P,T).
16 :- disk(D), not 1 { on(D,P,T) } 1, T=1..n.

18 #show move/3.

20 _minimize(1,T) :- ngoal(T).
```

Listing 28. Bounded towers of hanoi encoding (`tohB.lp`)

```
1 peg(a;b;c).
2 disk(1..4).
3 init_on(1..4,a).
4 goal_on(1..4,c).
```

Listing 29. Towers of hanoi instance (`tohI.lp`)

solve the problem in the minimum number of steps within a given bound, the goal is tested at each time step in Line 3. Once it is established, we do not permit any further moves and the goal persists in the following steps. This allows us to read off whether the goal was reached at the planning horizon (in Line 4). The state transition function along with state constraints are described in Lines 6–16. Since the encoding of the Towers of Hanoi problem is fairly standard, we focus in the sequel on implementing branch-and-bound-based minimization. In view of this, note that Line 6 ensures that moves are only permitted if the goal is not yet achieved in the previous state. Thus, the subsequent states do not change anymore, which allows us to express the optimization function in Line 20 as minimizing the number of steps in which the goal is not reached.

The idea of branch-and-bound-based optimization is to compute an optimal solution by producing a series of increasingly better solutions until no better solution is found. The solution obtained last is then an optimal one. Listing 30 implements the corresponding optimization algorithm via *clingo*'s `Application` class from Section 4. It starts by overriding *clingo*'s `main` function in Line 20 and begins with reading the input either from `files` provided on the command line or from standard input in Lines 21 and 24.

The basic building block of our algorithm consists of a weight constraint embedded in the following subprogram.

```
1 #program bound(b).
2 :- #sum { V,I: _minimize(V,I) } >= b.
```

```
1  import sys
2  from clingo.symbol import Number, SymbolType
3  from clingo.application import Application, clingo_main

5  class OptExampleApp(Application):
6      program_name = "opt-example"
7      version = "1.0"

9      def __init__(self):
10         self._bound = None

12     def _on_model(self, model):
13         self._bound = 0
14         for atom in model.symbols(atoms=True):
15             if (atom.match("_minimize", 2)
16             and atom.arguments[0].type
17             is  SymbolType.Number):
18                 self._bound += atom.arguments[0].number

20     def main(self, ctl, files):
21         if not files:
22             files = ["-"]
23         for f in files:
24             ctl.load(f)
25         ctl.add("bound", ["b"],
26             ":- #sum { V,I: _minimize(V,I) } >= b.")

28         ctl.ground([("base", [])])
29         while ctl.solve(on_model=self._on_model).satisfiable:
30             print("Found new bound: {}".format(self._bound))
31             ctl.ground([("bound", [Number(self._bound)])])

33         if self._bound is not None:
34             print("Optimum found")

36 clingo_main(OptExampleApp(), sys.argv[1:])
```

Listing 30. Branch-and-bound optimization (`opt.py`)

This program ensures that the next stable model yields a better bound than the one of the solution at hand. More precisely, it expects a bound `b` as parameter and adds the integrity constraint in Line 2 to enforce a better solution. A new instance of this program is added for each consecutive stable model. The addition of the (non-ground) constraint in Line 2 as part of program `bound(b)` is accomplished in Lines 25 and 26.

The actual minimization algorithm starts by grounding the `base` program in Line 28 before it enters the loop in Lines 29–31. This loop implements the branch-and-bound-based search for the minimum by searching for stable models while updating the bound until the problem is unsatisfiable. Note that we pass a callback to the `solve` function in Line 29. With it, the `_on_model` function in Lines 12–18 is called for every model found. If there is a stable model, Lines 14–18 iterate over the atoms of the stable model while summing up the current bound by extracting the weight of atoms over predicates `_minimize/2`.

We check that the first argument of the atom is an integer and ignore atoms where this is not the case; just like the `#sum` aggregate in Line 26. When a model was found, the body of the loop in Lines 29–31 is processed. First, the algorithm prints the bound in Line 30. Then, it adds an integrity constraint in Line 31 making sure that the next stable model is strictly better than the current one. Finally, if the program becomes unsatisfiable, the branch-and-bound loop in Lines 29–31 ends and Lines 33–34 print that the previously found stable model (if any) is the optimal solution.

When running the augmented logic program in Listings 28, 29, and 30 with a horizon of 17, the solver finds plans of length 17, 16, and 15 and shows that no plan of length 14 exists. This is reflected by *clingo*'s output indicating four solver calls and three stable models:

```
UNIX> python opt.py tohB.lp tohI.lp -c n=17
opt-example version 1.0
Reading from tohB.lp ...
Solving...
Answer: 1
move(4,b,1)  move(3,c,2)  move(4,a,3)  move(4,c,4)  move(2,b,5)  \
move(4,a,6)  move(3,b,7)  move(4,c,8)  move(4,b,9)  move(1,c,10) \
move(4,c,11) move(3,a,12) move(4,a,13) move(2,c,14) move(4,b,15)
move(3,c,16) move(4,c,17)
Found new bound: 17
Solving...
Answer: 1
move(4,b,1)  move(3,c,2)  move(4,c,3)  move(2,b,4)  move(4,a,5)  \
move(3,b,6)  move(4,c,7)  move(4,b,8)  move(1,c,9)  move(4,c,10) \
move(3,a,11) move(4,a,12) move(2,c,13) move(4,b,14) move(3,c,15)
move(4,c,16)
Found new bound: 16
Solving...
Answer: 1
move(4,b,1)  move(3,c,2)  move(4,c,3)  move(2,b,4)  move(4,a,5)  \
move(3,b,6)  move(4,b,7)  move(1,c,8)  move(4,c,9)  move(3,a,10) \
move(4,a,11) move(2,c,12) move(4,b,13) move(3,c,14) move(4,c,15)
Found new bound: 15
Solving...
Optimum found
UNSATISFIABLE
```

Note that at the end the solver deals with the ground program obtained from Listings 28 and 29 grounded with parameter `n=17` along with the three integrity constraints obtained from subprograms `bound(17)`, `bound(16)`, and `bound(15)`.

Last but not least, note that the functionality implemented above is equivalent to using *clingo*'s inbuilt optimization mode by replacing Line 20 in Listing 28 with

```
20 #minimize { 1,T: ngoal(T) }.
```

5.3 Incremental ASP solving

Incremental ASP solving offers a step-oriented approach to ASP that avoids redundancies by gradually processing the extensions to a problem rather than repeatedly re-processing the entire growing problem. To this end, a program is partitioned into a base part, describing static knowledge independent of the step parameter `t`, a cumulative part, capturing knowledge accumulating with increasing `t`, and a volatile part specific for each value of `t`. In *clingo*, all three parts are captured by `#program` declarations along with `#external` atoms for handling volatile rules, namely, subprograms named `base`, `step`,

```
1  #program base.
2  on(D,P,0) :- init_on(D,P).

4  #program check(t).
5  :- goal_on(D,P), not on(D,P,t), query(t).

7  #program step(t).
8  1 { move(D,P,t): disk(D), peg(P) } 1.

10 move(D,t)        :- move(D,P,t).
11 on(D,P,t)        :- move(D,P,t).
12 on(D,P,t)        :- on(D,P,t-1), not move(D,t).
13 blocked(D-1,P,t) :- on(D,P,t-1).
14 blocked(D-1,P,t) :- blocked(D,P,t), disk(D).

16 :- move(D,P,t), blocked(D-1,P,t).
17 :- move(D,t), on(D,P,t-1), blocked(D,P,t).
18 :- disk(D), not 1 { on(D,P,t) } 1.

20 #show move/3.
```

Listing 31. Towers of hanoi incremental encoding (`tohE.lp`)

and `check` along with external atoms of form `query(t)`. Note that these names have no general, predefined meaning; their composition is usually defined in an associated script.

We illustrate this by adapting the Towers of Hanoi encoding from Listing 28 to an incremental version in Listing 31. To this end, we arrange the original encoding in program parts `base`, `check(t)`, and `step(t)`, use `t` instead of `T` as time parameter, and simplify testing the goal. Checking the goal is easier here because the incremental approach guarantees a shortest plan and, hence, does not require additional minimization.

At first, we observe that the problem instance in Listing 29 as well as Line 2 in Listing 31 constitute static knowledge and thus belong to the `base` program. More interestingly, the query is expressed in Line 5 of Listing 31. Its volatility is realized by making it subject to the truth assignment to the external atom `query(t)`. For convenience, this atom is predefined in Line 38 in Listing 32 as part of the `check` program. Hence, for illustration, subprogram `check` consists of a user- and a pre-defined part. Finally, the transition function along with state constraints are described in the subprogram `step` in Lines 7–18.

The idea is now to control the successive grounding and solving of the program parts in Listings 29 and 31 by the Python script in Listing 32.[10] To this end, we use five variables to govern the loop in Lines 44–60.[11] Variables `imin` and `imax` prescribe a minimum and maximum number of iterations, respectively; `istop` gives a termination criterion, e.g., `"SAT"` or `"UNSAT"`. The value of `step` is used to instantiate the parametrized subprograms and `ret` gives the solving result. While the initial values of `step` and `ret` are set in Line 43, the first three variables are user-defined. We show how such user-defined variables are

[10] For brevity, we have stripped class `IncConfig` in Line 9 from the parsing methods `parse_int` and `parse_stop`. The full source code is available online (Potassco Team 2021h).

[11] This follows the original implementation of incremental ASP solving in *iclingo* (Gebser et al. 2008)

```
import sys
from clingo.symbol import Function, Number
from clingo.application import Application, clingo_main

class IncConfig:
    def __init__(self):
        self.imin, self.imax, self.istop = 1, None, "SAT"

[...]

class IncExampleApp(Application):
    program_name = "inc-example"
    version = "1.0"

    def __init__(self):
        self._conf = IncConfig()

    def register_options(self, options):
        group = "Inc-Example Options"
        options.add(
            group, "imin",
            f"Minimum number of steps [{self._conf.imin}]",
            parse_int(self._conf, "imin", min_val=0),
            argument="<n>")
        options.add(
            group, "imax",
            f"Maximum number of steps [{self._conf.imax}]",
            parse_int(self._conf, "imax", min_val=0, optional=True),
            argument="<n>")
        options.add(
            group, "istop",
            f"Stop criterion [{self._conf.istop}]",
            parse_stop(self._conf, "istop"))

    def main(self, ctl, files):
        if not files: files = ["-"]
        for f in files: ctl.load(f)
        ctl.add("check", ["t"], "#external query(t).")

        conf = self._conf
        imin, imax, istop = conf.imin, conf.imax, conf.istop

        step, ret = 0, None
        while ((imax is None or step < imax) and
               (step == 0 or step < imin or (
                   (istop == "SAT"     and not ret.satisfiable) or
                   (istop == "UNSAT"   and not ret.unsatisfiable) or
                   (istop == "UNKNOWN" and not ret.unknown)))):
            parts = []
            parts.append(("check", [Number(step)]))
            if step > 0:
                query = Function("query", [Number(step - 1)])
                ctl.release_external(query)
                parts.append(("step", [Number(step)]))
            else:
                parts.append(("base", []))
            ctl.ground(parts)
            query = Function("query", [Number(step)])
            ctl.assign_external(query, True)
            ret, step = ctl.solve(), step + 1

clingo_main(IncExampleApp(), sys.argv[1:])
```

Listing 32. Python script implementing incremental ASP solving in *clingo* (`inc.py`)

integrated into *clingo*'s option handling below, but first describe the actual implementation of incremental ASP solving.

The subprograms grounded in each iteration are accumulated in the list `parts` (cf. Line 49). Each of its entries is a pair consisting of a subprogram name along with its list of actual parameters. In the very first iteration, the subprograms `base` and `check(0)` are grounded. Note that this involves the declaration of the external atom `query(0)` and the assignment of its default value false. The latter is changed in Line 59 to true in order to activate the actual query. The `solve` call in Line 60 then amounts to checking whether the goal situation is already satisfied in the initial state. As well, the value of `step` is incremented to `1`.

As long as the termination condition remains unfulfilled, each following iteration takes the respective value of variable `step` to replace the parameter in subprograms `step` and `check` during grounding. In addition, the current external atom `query(t)` is set to true, while the previous one is permanently set to false. This disables the corresponding instance of the integrity constraint in Line 5 of Listing 31 before it is replaced in the next iteration. In this way, the query condition only applies to the current horizon.

In our example, the `solver` is called 16 times before a plan of length 15 is found:

```
UNIX> python inc.py tohE.lp tohI.lp
inc-example version 1.0
Reading from tohE.lp ...
Solving...
[...]
Solving...
Answer: 1
move(4,b,1)  move(3,c,2)  move(4,c,3)  move(2,b,4)  \
move(4,a,5)  move(3,b,6)  move(4,b,7)  move(1,c,8)  \
move(4,c,9)  move(3,a,10) move(4,a,11) move(2,c,12) \
move(4,b,13) move(3,c,14) move(4,c,15)
SATISFIABLE

Models       : 1+
Calls        : 16
```

Last but not least, let us explain how option processing can be added by sketching how the three variables `imin`, `imax`, and `istop` can be set from the command line. For this purpose, *clingo*'s API offers the two methods `register_options` and `validate_options`. In our simple example, we only use the former since the latter is meant to handle situations with conflicting options.[12] In fact, `register_options` receives an `ApplicationOptions` object as parameter to register additional options. For example, option `--imin` is added in Lines 20–24 by calling `ApplicationOptions.add`. Its first parameter is the name of an option group, used as the section heading when printing the application's help when called with `--help`. In our example, all options are grouped in the "`Inc-Example Options`" section. More concretely, we get the following help output (omitting descriptions for *clingo*'s options):

```
UNIX> python inc.py --help
[...]
Inc-Example Options:

  --imin=<n>              : Minimum number of steps [1]
  --imax=<n>              : Maximum number of steps [None]
```

[12] We also omit describing the underlying option parsers; cf. Footnote 10.

```
  --istop=<arg>           : Stop criterion [SAT]

[...]
```

The second parameter is the name of the option on the command line. Here, we pass `"imin"` to add option `--imin`. This is followed by the option description and the (omitted) option parser. Registration and parsing of options happens once before the `IncExampleApp.main` method is called. Hence, at that time the attributes of the applications `IncExampleApp._conf` object either have their default values or were overwritten by options passed on the command line. The last keyword parameter configures the placeholder, which is printed in the help output. Since we are adding a numeric option, it is good practice to call it `<n>`. By default it is called `<arg>`.

6 Theory-enhanced ASP solving

This section provides fundamental concepts for extending *clingo* with foreign types of constraints, also referred to as theories. To begin with, it is important to bear in mind that ASP is a modeling-grounding-solving paradigm, in contrast to, for instance, the solving-centered approach of SAT Modulo Theories (SMT; Nieuwenhuis et al. 2006). Hence extensions of ASP are rarely limited to single components but rather spread throughout the whole workflow. This begins with the addition of new language constructs to the input language, requiring in turn enhancements to the grounder as well as syntactic means for passing the ground constructs to a downstream system. In case the latter is an ASP solver, it must be enabled to handle the specific input and incorporate corresponding solving capacities. Finally, each such extension is rather specific and thus requires different means at all ends.

We begin by showing how *clingo*'s input language can be customized with theory-specific constructs. We then outline *clingo*'s algorithmic approach to ASP solving with theory propagation to put the description of *clingo*'s theory reasoning interface on firm grounds.

6.1 Input language

We begin by introducing the theory-related features of *clingo*'s input language. They are situated in the underlying grounder *gringo* and can thus also be used independently of *clingo*. We start with a detailed description of the generic means for defining theories and complement this in Appendix B with an overview of the corresponding intermediate language *aspif*.

The generic approach to theory specification rests upon two languages: the one defining theory languages and the theory language itself. Both borrow elements from the underlying ASP language, foremost an aggregate-like syntax for formulating variable length expressions. To illustrate this, consider Listing 33, where a logic program is extended by constructs for handling difference and linear constraints. While the former are binary constraints of the form $x_1 - x_2 \leq k$, the latter have a variable size and are of the form $a_1x_1 + \cdots + a_nx_n \circ k$, where x_i are integer variables, a_i and k are integers, and $\circ \in \{\leq, \geq, <, >, =\}$ for $1 \leq i \leq n$. Note that solving difference constraints is polynomial, while solving linear equations (over integers) is NP-complete. The theory language for expressing both types of constraints is defined in Lines 1–15 and preceded by the directive

```
1  #theory lc {
2    constant    { - : 0, unary };
3    diff_term   { - : 0, binary, left };
4    linear_term { + : 2, unary; - : 2, unary;
5                  * : 1, binary, left;
6                  + : 0, binary, left;
7                  - : 0, binary, left };
8    domain_term { .. : 1, binary, left };
9    show_term   { / : 1, binary, left };

11   &dom/0 : domain_term, {=}, linear_term, any;
12   &sum/0 : linear_term, {<=,=,>=,<,>,!=}, linear_term, any;
13   &diff/0 : diff_term, {<=}, constant, any;
14   &show/0 : show_term, directive
15 }.

17 #const n=2.  #const m=1000.

19 task(1..n).
20 duration(T,200*T) :- task(T).

22 &dom  { 1..m } = start(T) :- task(T).
23 &dom  { 1..m } = end(T)   :- task(T).
24 &diff { end(T)-start(T) } <= D :- duration(T,D).
25 &sum  { end(T): task(T); -start(T): task(T) } <= m.

27 &show { start/1; end/1 }.
```

Listing 33. Logic program enhanced with difference and linear constraints (`lc.lp`)

`#theory`. The elements of the resulting theory language are preceded by `&` and used as regular atoms in the logic program in Lines 17–27.

To be more precise, a *theory definition* has the form

`#theory` T `{`D_1`;`$\ldots$`;`D_n`}.`

where T is the theory name and each D_i is a definition for a theory term or a theory atom for $1 \leq i \leq n$. The language induced by a theory definition is the set of all theory atoms constructible from its theory atom definitions.

A *theory atom definition* has the form

`&`p`/`k `:` t`,`o or `&`p`/`k `:` t`,{`$\diamond_1$`,`$\ldots$`,`$\diamond_m$`},`t'`,`o

where p is a predicate symbol and k its arity, t and t' are names of theory term definitions, each $\diamond_i$ is a theory operator for $m \geq 1$, and $o \in \{\texttt{head}, \texttt{body}, \texttt{any}, \texttt{directive}\}$ determines where theory atoms may occur in a rule. Examples of theory atom definitions are given in Lines 11–14 of Listing 33. The language of a theory atom definition as given above contains all *theory atoms* of the form

`&`a `{`C_1`:`L_1`;`$\ldots$`;`C_n`:`L_n`}` or `&`a `{`C_1`:`L_1`;`$\ldots$`;`C_n`:`L_n`}` $\diamond$ c

where a is an atom over predicate p of arity k, each C_i is a tuple of theory terms in the language for t, c is a theory term in the language for t', $\diamond$ is a theory operator among $\{\diamond_1, \ldots, \diamond_m\}$, and each L_i is a regular condition (i.e., a tuple of regular literals) for $1 \leq i \leq n$. Whether the last part '$\diamond\, c$' is included depends on the form of a theory atom

definition. Further, observe that theory atoms with occurrence type `any` can be used both in the head and body of a rule; with occurrence types `head` and `body`, their usage can be restricted to rule heads and bodies only. Occurrence type `directive` is similar to type `head` but additionally requires that the rule body must be completely evaluated during grounding. Five occurrences of theory atoms can be found in Lines 22–27 of Listing 33.

Remark 6
Having conditions, such as L_i above, is useful to address variable length constraints as in Listing 33. However, atoms like `task(T)` are given as facts and therefore are not subject to solving. In general, however, conditions may involve atoms that are not decided during grounding. If this is the case, they have to be handled with care since there is no predefined behavior. For instance, a rule `a :- &sum{ x : a } >= 0` may be unsatisfiable under certain semantic principles (Gelfond and Zhang 2019). Formal foundations and implementation techniques of conditional theory atoms were developed by Cabalar et al. (2020a) and Cabalar et al. (2020b).

A *theory term definition* has the form

t `{`D_1`;`$\ldots$`;`D_n`}`

where t is a name for the defined terms and each D_i is a theory operator definition for $1 \leq i \leq n$. A corresponding definition specifies the language of all theory terms that can be constructed via its operators. Examples of theory term definitions are given in Lines 2–9 of Listing 33. Each resulting *theory term* is one of the following:

- a constant term: c
- a variable term: v
- a binary theory term: $t_1 \diamond t_2$
- a unary theory term: $\diamond t_1$
- a function theory term: $f(t_1, \ldots, t_k)$
- a tuple theory term: $(t_1, \ldots, t_l,)$
- a set theory term: $\{t_1, \ldots, t_l\}$
- a list theory term: $[t_1, \ldots, t_l]$

where each t_i is a theory term, $\diamond$ is a theory operator defined by some D_i, c and f are symbolic constants, v is a first-order variable, $k \geq 1$, and $l \geq 0$. (The trailing comma in tuple theory terms is optional if $l \neq 1$.) Parentheses can be used to specify operator precedence.

A *theory operator definition* has the form

$\diamond$ `:` p`,unary` or $\diamond$ `:` p`,binary,`a

where $\diamond$ is a unary or binary theory operator with precedence $p \geq 0$ (determining implicit parentheses). Binary theory operators are additionally characterized by an associativity $a \in \{\texttt{right}, \texttt{left}\}$. As an example, consider Lines 4–5 of Listing 33, where the `left` associative `binary` operators `+` and `*` are defined with precedence `2` and `1`, respectively. Hence, parentheses in terms like '`(X+(2*Y))+Z`' can be omitted. In total, Lines 2–9 of Listing 33 include nine theory operator definitions. Specific *theory operators* can be assembled (written consecutively without spaces) from the symbols '`!`', '`<`', '`=`', '`>`', '`+`', '`-`', '`*`', '`/`', '`\`', '`?`', '`&`', '`|`', '`.`', '`:`', '`;`', '`~`', and '`^`'. For instance, in Line 8 of Listing 33, the operator '`..`' is defined as the concatenation of two periods. The tokens '`.`', '`:`', '`;`', and '`:-`' must be combined with other symbols due to their dedicated usage. Instead, one may write '`..`', '`::`', '`;;`', '`::-`', etc.

While theory terms are formed similar to regular ones, theory atoms rely upon an aggregate-like construction for forming variable-length theory expressions. In this way,

```
 1  task(1).
 2  task(2).
 3  duration(1,200).
 4  duration(2,400).

 6  &dom { 1..1000 } = start(1).
 7  &dom { 1..1000 } = start(2).
 8  &dom { 1..1000 } = end(1).
 9  &dom { 1..1000 } = end(2).

11  &diff { end(1)-start(1) } <= 200.
12  &diff { end(2)-start(2) } <= 400.

14  &sum { end(1); end(2); -start(1); -start(2) } <= 1000.

16  &show { start/1; end/1 }.
```

Listing 34. Human-readable result of grounding Listing 33 via '`gringo --text lc.lp`'

standard grounding techniques can be used for gathering theory terms. The treatment of theory terms still differs from their regular counterparts in that the grounder skips simplifications like, e.g., arithmetic evaluation. This can be seen on the different results in Listing 34 of grounding terms formed with the regular and theory-specific variants of operator '`..`'. Observe that the fact `task(1..n)` in Line 19 of Listing 33 results in `n` ground facts, viz. `task(1)` and `task(2)` because of `n=2`. Unlike this, the theory expression `1..m` stays structurally intact and is only transformed into `1..1000` in view of `m=1000`. That is, the grounder does not evaluate the theory term `1..1000` and leaves its interpretation to a downstream theory solver. A similar situation is encountered when comparing the treatment of the regular term '`200*T`' in Line 20 of Listing 33 to the theory term '`end(T)-start(T)`' in Line 24. While each instance of '`200*T`' is evaluated during grounding, instances of the theory term '`end(T)-start(T)`' are left intact in Lines 11 and 12 of Listing 34. In fact, if '`200*T`' had been a theory term as well, it would have resulted in the unevaluated instances '`200*1`' and '`200*2`'.

6.2 Semantic principles

Given the hands-on nature of this work, we only give an informal idea of the semantic principles underlying theory solving in ASP.

A logic program induces a set of stable models. To extend this concept to logic programs with theory expressions, we follow the approach of lazy theory solving (Barrett et al. 2009). We abstract from the specific semantics of a theory by considering the theory atoms representing the underlying theory constraints. The idea is that a regular stable model of a program over regular and theory atoms is only valid with respect to a theory, if the constraints induced by the truth assignment to the theory atoms are satisfiable in the theory.

In the example above, this amounts to finding a numeric assignment to all theory variables satisfying all difference and linear constraints associated with theory atoms. The ground program in Listing 34 has a single stable model consisting of all regular and theory

atoms in Lines 1–16. Here, we easily find assignments satisfying the induced constraints, e.g., `start(1)` $\mapsto 1$, `end(1)` $\mapsto 2$, `start(2)` $\mapsto 2$, and `end(2)` $\mapsto 3$.

In fact, there are alternative semantic options for capturing theory atoms, as discussed by Gebser et al. (2016). First of all, we may distinguish whether imposed constraints are only determined outside or additionally inside a logic program. This leads to the distinction between *defined* and *external* theory atoms (analogous to rule heads and input atoms defined by `#external` directives). While external theory atoms must only be satisfied by the respective theory, defined ones must additionally be derivable through rules in the program. A second distinction concerns the interplay of ASP with theories. More precisely, it is about the logical correspondence between theory atoms and theory constraints. This leads us to the distinction between *strict* and *non-strict* theory atoms. The strict correspondence requires a constraint to be satisfied *iff* the associated theory atom is true. A weaker since only implicative condition is imposed in the non-strict case. Here, a constraint must hold *only if* the associated theory atom is true. In other words, only non-strict theory atoms assigned true impose requirements, while constraints associated with falsified non-strict theory atoms are free to hold or not. However, by contraposition, a violated constraint leads to a false non-strict theory atom.

6.3 Algorithmic aspects

The algorithmic approach to ASP solving modulo theories of *clingo*, or more precisely that of its underlying ASP solver *clasp*, follows the lazy approach to solving in Satisfiability Modulo Theories (SMT; Barrett et al. 2009). We give below an abstract overview that serves as light algorithmic underpinning for the description of *clingo*'s implementation given in the next section.

A ground program P induces *completion* and *loop nogoods*, called Δ_P or Λ_P, respectively, that can be used for computing stable models of P (Gebser et al. 2012). Nogoods represent invalid partial assignments and can be thought of as negative Boolean constraints. We represent (partial) assignments as consistent sets of literals. An assignment is total if it contains either the positive or negative literal of each atom. We say that a nogood is violated by an assignment if the former is contained in the latter; a nogood is unit if all but one of its literals are in the assignment. Each total assignment not violating any nogood in $\Delta_P \cup \Lambda_P$ yields a regular stable model of P, and such an assignment is called a solution (for $\Delta_P \cup \Lambda_P$). To accommodate theories, we identify a theory T with a set Δ_T of *theory nogoods*, and extend the concept of a solution in the straightforward way.

The nogoods in $\Delta_P \cup \Lambda_P \cup \Delta_T$ provide the logical foundation for the Conflict-Driven Constraint Learning (CDCL) procedure (Marques-Silva et al. 2009; Gebser et al. 2012) outlined in Figure 2. While the completion nogoods in Δ_P are usually made explicit and subject to unit propagation,[13] the loop nogoods in Λ_P as well as theory nogoods in Δ_T are typically handled by dedicated propagators and particular members are selectively recorded. While a dedicated propagator for loop nogoods is built-in in systems like *clingo*, those for theories are provided via the interface **Propagator** in Figure 3. To utilize custom propagators, the algorithm in Figure 2 includes an *initialization* step in Line (I).

[13] Unit propagation extends an assignment with literals complementary to the ones missing in unit nogoods.

```
(I) initialize                                  // register theory propagators and initialize watches
    loop
        propagate completion, loop, and recorded nogoods    // deterministically assign literals
        if no conflict then
            if all variables assigned then
(C)             if some δ ∈ Δ_T is violated then record δ       // theory propagators check Δ_T
                else return variable assignment                 // theory-based stable model found
            else
(P)             propagate theories  // theory propagators may record theory nogoods from Δ_T
                if no nogood recorded then decide  // non-deterministically assign some literal
        else
            if top-level conflict then return unsatisfiable
            else
                analyze                          // resolve conflict and record a conflict constraint
(U)             backjump                        // undo assignments until conflict constraint is unit
```

Fig. 2. Basic algorithm for Conflict-Driven Constraint Learning (CDCL) modulo theories

In addition to the "registration" of a propagator for a theory as an extension of the basic CDCL procedure, common tasks performed in this step include setting up internal data structures and so-called watches for (a subset of) the theory atoms, so that the propagator is invoked (only) when some watched literal gets assigned.

As usual, the main CDCL loop starts with unit propagation on completion and loop nogoods, the latter handled by the respective built-in propagator, as well as any nogoods already recorded. If this results in a non-total assignment without conflict, theory propagators for which some of their watched literals have been assigned are invoked in Line (P). A propagator for a theory T can then inspect the current assignment, update its data structures accordingly, and most importantly, perform *theory propagation* determining theory nogoods $\delta \in \Delta_T$ to record. Usually, any such nogood δ is unit or conflicting in order to trigger unit propagation or conflict resolution, although this is not a necessary condition. The interplay of unit and theory propagation continues until a conflict or a total assignment arises, or no (further) watched literals of theory propagators get assigned by unit propagation. In the latter case, some non-deterministic decision is made to extend the partial assignment at hand and then to proceed with unit and theory propagation.

If no conflict arises and an assignment is total, in Line (C), theory propagators are called, one by one, for a final *check*. The idea is that, e.g., a "lazy" propagator for a theory T that does not exhaustively test violations of its theory nogoods by partial assignments can make sure that the assignment is indeed a solution for Δ_T, or record some violated nogood(s) from Δ_T otherwise. Even in case theory propagation on partial assignments is exhaustive and a final check is not needed to detect conflicts, the information that search led to a total assignment can be useful in practice, e.g., to store values for integer variables like `start(1)`, `start(2)`, `end(1)`, and `end(2)` in Listing 34 that witness the existence of a solution for T.

Finally, in case of a conflict, i.e., some completion or recorded nogood is violated by the current assignment, provided that some non-deterministic decision is involved in the conflict, a new conflict constraint is recorded and utilized to guide backjumping in Line (U), as usual with CDCL. In a similar fashion as the assignment of watched

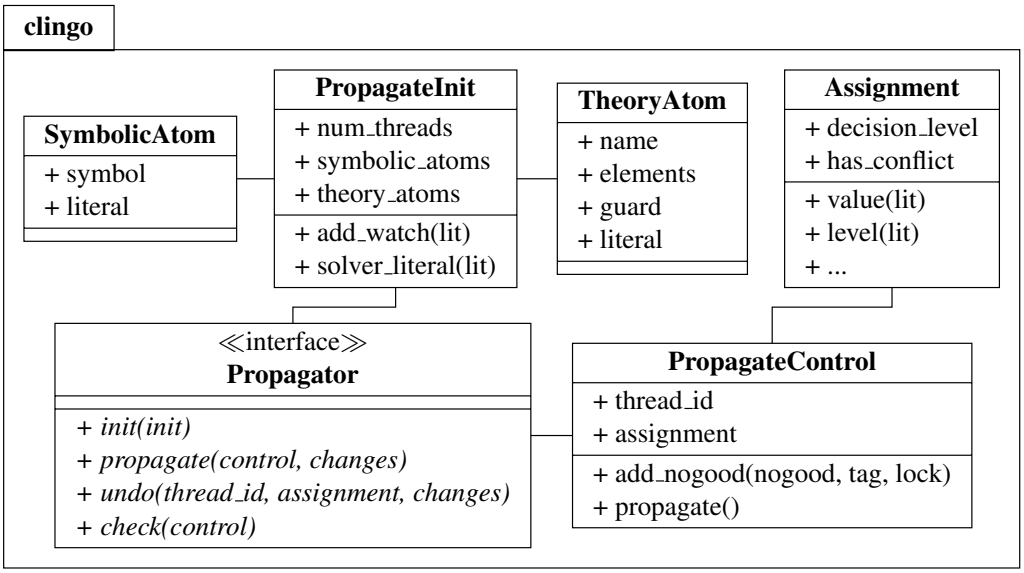


Fig. 3. Class diagram of *clingo*'s (theory) propagator interface

literals serves as trigger for theory propagation, theory propagators are informed when they become unassigned upon backjumping. This allows the propagators to *undo* earlier operations, e.g., internal data structures can be reset to return to a state taken prior to the assignment of watches.

In summary, the basic CDCL procedure is extended in four places to account for custom propagators: initialization, propagation of (partial) assignments, final check of total assignments, and undo steps upon backjumping.

6.4 Propagator interface

We now turn to the implementation of theory propagation in *clingo* and detail the structure of its interface depicted in Figure 3. The interface `Propagator` has to be implemented by each custom propagator. After registering such a propagator with *clingo*, its functions are called during initialization and search as indicated in Figure 2. Function `init` is called once before solving (Line (I) in Figure 2) to allow for initializing data structures used during theory propagation. It is invoked with a `PropagateInit` object providing access to symbolic (`SymbolicAtom`) as well as theory (`TheoryAtom`) atoms. Both kinds of atoms are associated with program literals, which are in turn associated with solver literals. Program as well as solver literals are identified by non-zero integers, where positive and negative numbers represent positive or negative literals, respectively. In order to get notified about assignment changes, a propagator can set up watches on solver literals during initialization.

During search, function `propagate` is called with a `PropagateControl` object and a (non-empty) list of watched literals that got assigned in the recent round of unit propagation (Line (P) in Figure 2). The `PropagateControl` object can be used to inspect the current assignment, record nogoods, and trigger unit propagation. Furthermore, to

support multi-threaded solving, its `thread_id` property identifies the currently active thread, each of which can be viewed as an independent instance of the CDCL algorithm in Figure 2.[14] Function `undo` is the counterpart of `propagate` and called whenever the solver retracts assignments to watched literals (Line (U) in Figure 2). In addition to the list of watched literals that have been retracted (in chronological order), it receives the identifier and the assignment of the active thread. Finally, function `check` is similar to `propagate`, yet invoked without a list of changes. Instead, it is (only) called on total assignments (Line (C) in Figure 2), independently of watches. Overriding the empty default implementations of propagator methods is optional.

7 Extending ASP with difference constraints

In this section, we develop a case-study featuring the extension of ASP with difference constraints[15] by augmenting *clingo* with a corresponding propagator. To this end, we extend the language of Section 2 with difference constraint atoms of form

`&diff { `u`-`v` } <= `d

where u and v are (regular) terms and d is an integer constant. Such atoms may either occur in the head or the body of a rule. Hence, stable models may now also include theory atoms of form '`&diff { `u`-`v` } <= `d'. More precisely, for a stable model X, let C_X be the set of *difference constraints* $u-v \leq d$ associated with theory atoms '`&diff { `u`-`v` } <= `d' in X and V_X be the set of all (integer) variables occurring in the difference constraints in C_X. In our case, a stable model X is then *DC-stable*, if there is a mapping from V_X to the integers, first, satisfying all constraints in C_X, and second, falsifying all constraints not in C_X that are associated with a *strict* difference constraint atom (Janhunen et al. 2017).

Next, let us discuss the semantic principles guiding our implementation. Recall from Section 6.2 that theory atoms may have different semantic properties, either defined or external depending upon their occurrence, or either strict or non-strict depending upon their logical relation to the represented constraint. For difference constraints, the combinations of *strict* and *external* as well as *non-strict* and *defined* appear to be the most intuitive combinations (Janhunen et al. 2017).

To illustrate this, consider the following example:

```
1  &diff { 0-x } <= -2.
2  a :- &diff { 0-x } <= -1.
```

This program states that x is greater or equal 2 and that `a` is derived if x is greater or equal than 1. An intuitive result is to assign x a value greater or equal than 2 and to derive `a`. However, in case atom '`&diff { 0-x } <= -1`' is non-strict, we also obtain answer sets without `a`. This is because the falsity of '`&diff { 0-x } <= -1`' does not imply that $0-x \leq -1$ is false as well. This is enforced by interpreting the relation between '`&diff { 0-x } <= -1`' and $0-x \leq -1$ as strict, and then a is obtained. Intuitively, the combination of external and strict can be seen as interpreting theory atoms relative to an external oracle, according to which all possibilities have to be considered as the logic program is oblivious to the meaning of the theory atom.

[14] Depending on the configuration of *clasp*, threads can communicate with each other. For example, some of the recorded nogoods can be shared. This is transparent from the perspective of theory propagators.

[15] In SMT, the underlying formal system is also referred to as *quantifier free integer difference logic*.

```
10 THEORY = """
11 #theory dl{
12     diff_term {
13     -  : 3, unary;
14     ** : 2, binary, right;
15     *  : 1, binary, left;
16     /  : 1, binary, left;
17     \\ : 1, binary, left;
18     +  : 0, binary, left;
19     -  : 0, binary, left
20     };
21     &diff/1 : diff_term, {<=}, diff_term, any
22 }.
23 """
```

Listing 35. Theory language `dl` for difference constraints (`dl.py`, Lines 10–23)

For a complement to the above example, consider the following one:

```
1  &diff { 0-x } <= -2.
2  &diff { 0-x } <= -1 :- a.
```

Again, the program states that x is greater equal 2 but now atom `a` derives that x is greater or equal than 1. As we do not have a definition of `a`, we expect answer sets not containing `a` and assignments where x is greater or equal 2. However, once we interpret the atom '`&diff { 0-x } <= -1`' as strict, the program becomes unsatisfiable. In this case the falsity of '`&diff { 0-x } <= -1`' implies that $0 - x \leq -1$ is false as well. A non-strict interpretation avoids this and yields the expected result. The combination of non-strict and defined lets the logic program decide which theory atoms hold. Specifically, the absence of an atom in an answer set does not imply that the constraint is false but rather that it is not enforced. As a result, we handle occurrences of difference constraint atoms in the head as defined and non-strict, and atom occurrences in the body as external and strict.[16]

Let us now turn to the actual extension of *clingo*. The overall implementation is divided in two, on the one hand, the actual application class `DLApp` addressing grounding and solving in Listing 37, and on the other hand, six classes dealing with various aspects of difference constraints. The complete source code is available online (Potassco Team 2021d).

In what follows, we concentrate on the `HeadBodyTransformer` class in Listing 36, illustrating the manipulation of a logic program's abstract syntax tree (AST), as well as the `DLPropagator` class in Listing 38, showcasing a propagator adding foreign inferences to ASP. To support this, we also describe the interface of the `Graph` class but refrain from presenting its implementation. For expressing difference constraints, we define the theory language `dl` in Listing 35, a subset of the theory language `lc` presented in Listing 33 above. Note that to accommodate the additional argument indicating the location of the difference constraint atom, we have replaced `&diff/0` by `&diff/1`.

To achieve the above distinction between head and body occurrences of theory atoms

[16] Note that this amounts to treating occurrences of the same constraint atom in different ways. This is not unusual since the same constraint may be represented by syntactically different constraint atoms.

```
79 class HeadBodyTransformer(ast.Transformer):
80     def visit_Literal(self, lit, in_lit=False):
81         return lit.update(**self.visit_children(lit, True))

83     def visit_TheoryAtom(self, atom, in_lit=False):
84         term = atom.term
85         if term.name == "diff" and not term.arguments:
86             loc = "body" if in_lit else "head"
87             atom = atom.update(term=ast.Function(
88                 term.location, term.name,
89                 [ast.Function(term.location, loc, [], False)],
90                 False))
91         return atom
```

Listing 36. `HeadBodyTransformer` class for tagging occurrence of theory atoms (`dl.py`, Lines 79–91)

without changing the input language, we use *clingo*'s functionalities to modify the AST of non-ground programs for tagging theory atoms with their respective occurrence. Although pragmatic, the annotation of theory atoms has turned out to be very useful in several implementations. Moreover, it serves us as a first example of how non-ground programs can be modified through *clingo*'s API.

Remark 7
The user still writes difference constraints over `&diff/0`. The AST modification occurs on the non-ground level during parsing. Once grounded, it is checked whether all theory atoms are valid with regards to a theory language. At this point, difference constraints are constructed over `&diff/1`, which is opaque to the user.

As mentioned, this is accomplished by the `HeadBodyTransformer` class in Listing 36. This class nicely illustrates how the *visitor design pattern* is used by *clingo* to manipulate the AST of (non-ground) logic programs. The transformer uses the property that theory atoms in rule heads or bodies are never or always children of literal nodes, respectively. Assuming that the root node of the AST is visited with `in_lit=False`, function `visit_Literal` in Lines 80–81 visits its children with `in_lit=False`. Hence, function `visit_TheoryAtom` in Lines 83–91 is visited with the parameter `in_lit` indicating a head or body occurrence and returns the theory atom with the location (either `head` or `body`) as an argument. For instance, treating the example above using this class results in the following program:

```
1 &diff(head) { 0-x } <= -2.
2 a :- &diff(body) { 0-x } <= -1.
```

Listing 37 shows the application that addresses grounding and solving. Lines 16–25 implement a customized main function. The difference to *clingo*'s regular one is that a propagator for difference constraints is registered, the string variable `THEORY` containing the above theory language is added as a program, and the input programs are rewritten adding locations to the difference constraint atoms; grounding and solving then follow as usual. Note that the `solve` function in Line 25 takes a model callback as argument. Whenever a DC-stable model X is found, this callback adds symbols to the answer set representing a mapping satisfying the corresponding difference constraints C_X. The model X (excluding theory atoms) is printed as part of *clingo*'s default output. The callback

```
1  import sys
2  from clingo.application import Application, clingo_main
3  from clingo.ast import ProgramBuilder, parse_files
4  from dl import DLPropagator, HeadBodyTransformer, THEORY

6  class DLApp(Application):
7      program_name = "clingo-dl"
8      version = "1.0"

10     def __init__(self):
11         self._propagator = DLPropagator()

13     def on_model(self, model):
14         self._propagator.on_model(model)

16     def main(self, ctl, files):
17         ctl.register_propagator(self._propagator)
18         ctl.add("base", [], THEORY)

20         with ProgramBuilder(ctl) as bld:
21             hbt = HeadBodyTransformer()
22             parse_files(files, lambda stm: bld.add(hbt.visit(stm)))

24         ctl.ground([("base", [])])
25         ctl.solve(on_model=self.on_model)

27 sys.exit(int(clingo_main(DLApp(), sys.argv[1:])))
```

Listing 37. Application class `DLApp` with main loop for difference constraints (`dl-app.py`)

function `on_model` in Line 13 calls in turn the `on_model` function of the propagator (Line 13 in Listing 38) that adds symbols of the form $\texttt{dl}(x, v)$ to the model, where x is the name of an integer variable and v the assigned value in X.

Our example propagator for difference constraints in Listing 38 implements the algorithm presented by Cotton and Maler (2006). The idea is that deciding whether a set of difference constraints is satisfiable can be mapped to a graph problem. Given a set of difference constraints, let (V, E) be the weighted directed graph such that V is the set of variables occurring in the constraints and E the set of weighted edges (u, v, d) for each constraint $u - v \leq d$. The set of difference constraints is satisfiable if the corresponding graph does not contain a negative cycle (i.e. a cycle whose sum of edge labels is negative). The class

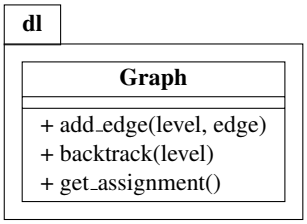


Fig. 4. Class diagram for the `Graph` class

```
192 class DLPropagator(Propagator):
193     def __init__(self):
194         self._l2e = {}    # {literal: [(node, node, weight)]}
195         self._e2l = {}    # {(node, node, weight): [literal]}
196         self._states = [] # [Graph]

198     def _state(self, thread_id):
199         while len(self._states) <= thread_id:
200             self._states.append(Graph())
201         return self._states[thread_id]

203     def _lit(self, control, edge):
204         for lit in self._e2l[edge]:
205             if control.assignment.is_true(lit):
206                 return lit
207         assert False

209     def _add_edge(self, init, lit, u, v, w):
210         edge = (u, v, w)
211         self._l2e.setdefault(lit, []).append(edge)
212         self._e2l.setdefault(edge, []).append(lit)
213         init.add_watch(lit)

215     def init(self, init):
216         for atom in init.theory_atoms:
217             term = atom.term
218             if term.name == "diff" and len(term.arguments) == 1:
219                 u = _eval(atom.elements[0].terms[0].arguments[0])
220                 v = _eval(atom.elements[0].terms[0].arguments[1])
221                 w = _eval(atom.guard[1]).number
222                 lit = init.solver_literal(atom.literal)
223                 self._add_edge(init, lit, u, v, w)
224                 if term.arguments[0].name == "body":
225                     self._add_edge(init, -lit, v, u, -w - 1)

227     def propagate(self, control, changes):
228         state = self._state(control.thread_id)
229         level = control.assignment.decision_level
230         for lit in changes:
231             for edge in self._l2e[lit]:
232                 cycle = state.add_edge(level, edge)
233                 if cycle is not None:
234                     c = [self._lit(control, e) for e in cycle]
235                     control.add_nogood(c) and control.propagate()
236                     return

238     def undo(self, thread_id, assign, changes):
239         self._state(thread_id).backtrack(assign.decision_level)

241     def on_model(self, model):
242         assignment = self._state(model.thread_id).get_assignment()
243         model.extend([Function("dl", [var, Number(value)])
244                       for var, value in assignment])
```

Listing 38. `DLPropagator` class for difference constraints (`dl.py`, Lines 192–244)

is in charge of cycle detection; its interface is given in Figure 4. We refrain from giving its code and rather concentrate on describing its interface:

- Function `add_edge` adds an edge of form `(u,v,d)` to the graph. If adding an edge to the graph leads to a negative cycle, the function returns the cycle in form of a list of edges; otherwise, it returns `None`. Furthermore, each edge added to the graph is associated with a decision level[17]. This additional information is used to backtrack to a previous state of the graph, whenever the solver has to backtrack to recover from a conflict.
- Function `backtrack` takes a decision level as argument. It removes all edges added on that level from the graph. For this to work, decision levels have to be backtracked in chronological order. Note that the CDCL algorithm in Figure 2 calling our propagator also backtracks decision levels in chronological order.
- The `Graph` class internally maintains an assignment of integers to nodes. This assignment can be turned into an assignment to the variables such that the difference constraints corresponding to the edges of the graph are satisfied. Function `get_assignment` returns this assignment in form of a list of pairs of variables and integers.

The difference logic propagator implements the `Propagator` interface (except for `check`) in Figure 3 in Lines 215–239; it features aspects like incremental propagation and backtracking, while supporting solving with multiple threads, and multi-shot solving. Whenever the set of edges associated with the current partial assignment of a solver induces a negative cycle and, hence, the corresponding difference constraints are unsatisfiable, it adds a nogood forbidding the negative cycle. To this end, it maintains data structures for detecting whether there is a conflict upon the addition of new edges. More precisely, the propagator has three data members:

1. The `self._l2e` dictionary in Line 194 maps solver literals for difference constraint theory atoms to their corresponding edges,[18]
2. the `self._e2l` dictionary in Line 195 maps edges back to solver literals,[19] and
3. the `self._states` list in Line 196 stores for each solver thread its current graph with the edges assigned so far.

Function `init` in Lines 215–225 sets up watches as well as the dictionaries `self._l2e` and `self._e2l`. To this end, it traverses the theory atoms over `diff`/1 in Lines 216–225. Note that the loop simply ignores other theory atoms treated by other propagators. In Lines 219–221, we extract the edge from the theory atom.[20] Each such atom is associated with a solver literal, obtained in Line 222. The mappings between solver literals and corresponding edges are then stored in the `self._l2e` and `self._e2l` dictionaries in

[17] The ASP solver's assignment comprises the decision level; it is incremented for each decision made and decremented for each decision undone while backjumping; initially, the decision level is zero.
[18] A solver literal might be associated with multiple edges.
[19] In one solving step, the *clingo* API guarantees that a (grounded) theory atom is associated with exactly one solver literal. Theory atoms grounded in later solving steps can be associated with fresh solver literals though.
[20] For brevity, we omit the definition of the `_eval` function, converting a theory term into a symbol. Furthermore, we assume that the user supplies valid theory atoms. A mature propagator checks validity and provides error messages.

Lines 211 and 212.[21] In Line 213 of the loop, a watch is added for each solver literal at hand, so that the solver calls `propagate` whenever the edge has to be added to the graph. Up to here, we accommodated the non-strict semantics as we only consider the constraint occurring in the program and not its negation. If the difference constraint atom occurs in the body, we impose the strict semantics, meaning that, in case that the assigned literal is false, we make sure that the negation of the difference constraint holds. We check if the atom occurs in the body in Line 224, and if this is the case, we add an edge representing the negation of the difference constraint associated with the negated literal and watch the negated literal as well.

Function `propagate`, given in Lines 227–236, accesses `control.thread_id` in Line 228 to obtain the graph associated with the active thread. The loops in Lines 230–236 then iterate over the list of changes and associated edges. In Line 232 each such edge is added to the graph. If adding the edge produces a negative cycle, a nogood is added in Line 235. Because an edge can be associated with multiple solver literals, we use function `_lit` retrieving the first solver literal associated with an edge that is true, to construct the nogood forbidding the cycle. Given that the solver has to resolve the conflict and backjump, the call to `add_nogood` always yields false, so that propagation is stopped without processing the remaining changes any further.[22]

Given that each edge added to the graph in Line 232 is associated with the current decision level, the implementation of function `undo` is quite simple. It calls function `backtrack` on the solver's graph to remove all edges added on the current decision level.

Remark 8
Here, we used a simplified Python version of the difference constraints propagator as a showcase. In practice, performance might fall short compared to solutions implemented in C or C++. The *clingo* package also offers the `clingo.theory` module to load propagators implemented in other languages via the *C Foreign Function Interface* for Python (CFFI 2021), thus combining convenient scripting with performance. The propagators of the extended ASP systems *clingo*[DL] and *clingcon* can be loaded using this interface and can be used as a basis to implement customized ASP systems.

7.1 Solving flow shop problems

To see our propagator in action, we consider the flow shop problem, dealing with a set of tasks T that have to be consecutively executed on m machines. Each task has to be processed on each machine from 1 to m. Different parts of one task are completed on each machine resulting in the completion of the task after execution on all machines is finished. Before a task can be processed on machine i, it has to be finished on machine $i-1$. The duration of different tasks on the same machine may vary. A task can only be executed on one machine at a time and a machine must not be occupied by more than

[21] Python's `setdefault` function is used to update the mappings. Depending on whether the given `key` already appears in the dictionary, the function either retrieves the associated value or inserts and returns the second argument.

[22] The optional arguments `tag` and `lock` of `add_nogood` can be used to control the scope and lifetime of recorded nogoods. Furthermore, if a propagator adds nogoods that are not necessarily violated, function `control.propagate` can be invoked to trigger unit propagation.

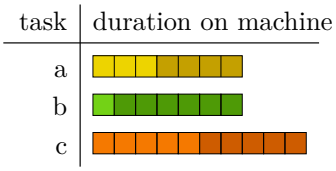


Fig. 5. Flow shop instance with three tasks and two machines

```
1                machine(1).       machine(2).
2  task(a). duration(a,1,3). duration(a,2,4).
3  task(b). duration(b,1,1). duration(b,2,6).
4  task(c). duration(c,1,5). duration(c,2,5).
```

Listing 39. Flow shop instance from Figure 5 (`fsI.lp`)

```
1  1 { cycle(T,U): task(U), U!=T } 1 :- task(T).
2  1 { cycle(T,U): task(T), U!=T } 1 :- task(U).
3  reach(M) :- M = #min { T: task(T) }.
4  reach(U) :- reach(T), cycle(T,U).
5  :- task(T), not reach(T).

7  1 { start(T): task(T) } 1.
8  permutation(T,U) :- cycle(T,U), not start(U).

10 seq((T,M),(T,M+1),D) :- task(T), duration(T,M,D), machine(M+1).
11 seq((T1,M),(T2,M),D) :- permutation(T1,T2), duration(T1,M,D).

13 &diff { T1-T2 } <= -D :- seq(T1,T2,D).
14 &diff { 0-(T,M) } <= 0 :- duration(T,M,D).

16 #show permutation/2.
```

Listing 40. Encoding of flow shop using difference constraints (`fsE.lp`)

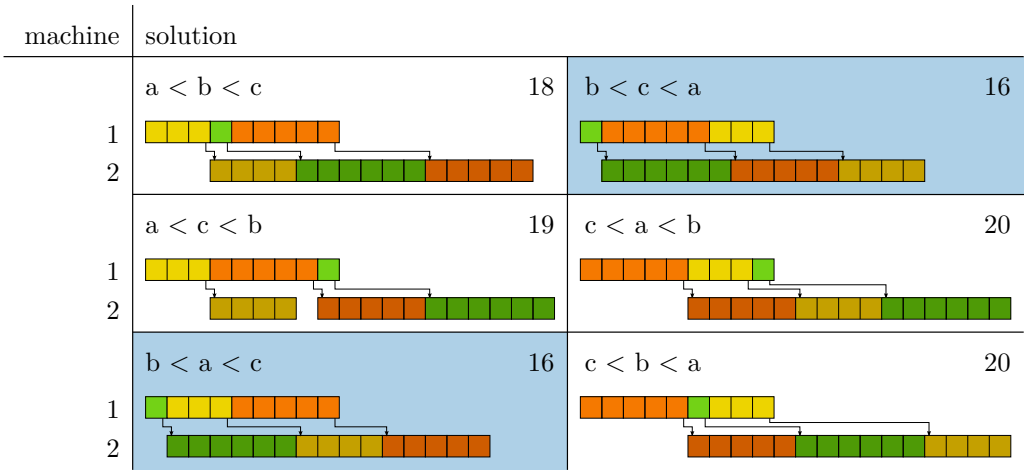


Fig. 6. Flow shop solutions for all possible permutations with the total execution length in the top right corner and optimal solutions with a blue background

one task at a time. An (optimal) solution to the problem is a permutation of tasks so that all tasks are finished as early as possible.

Figure 5 depicts a possible instance for the flow shop problem. The three tasks `a`, `b`, and `c` have to be scheduled on two machines. The colored boxes indicate how long a task has to run on a machine. Lighter shades of the same color are for the first and darker ones for the second machine. For example, task `a` needs to be processed for 3 time units on the first and 4 time units on the second machine.

Next, we encode this problem using ASP with difference constraints. We give in Listing 39 a straightforward encoding of the instance in Figure 5. Listing 40 provides the encoding of the flow shop problem. Following the generate, define, and test methodology of ASP (Lifschitz 2019), we first generate in Lines 1–8 all possible permutations of tasks, where atoms of the form `permutation(T,U)` encode that task T has to be executed before task U. Then, in the following Lines 10–14, we use difference constraints to calculate the duration of the generated permutation. The difference constraint in Line 13 guarantees that the tasks are executed in the right order. For example, `(a,1)` $-$ `(a,2)` $\leq -d$ ensures that task `a` can only be executed on machine `2` if it has finished on machine `1`. Hence, the variable `(a,2)` has to be assigned so that it is greater or equal to `(a,1)` $+ d$ where d is the duration of task `a` on machine `1`. Similarly, `(a,1)` $-$ `(b,1)` $\leq -d$ makes sure that task `b` can only be executed on machine `1` if task `a` has finished on machine `1`. While the first constraint results in a set of facts (see Line 10), the latter is subject to the generated permutation of tasks (see Line 11). The difference constraint in Line 14 ensures that all time points at which a task is started are greater than zero. Note that this constraint is in principle redundant but since sets of difference constraints always have infinitely many solutions it is good practice to encode relative to a starting point. Furthermore, note that `0` is actually a variable. In fact, the `Graph` class takes care of subtracting the value of variable `0` from all other variables when returning an assignment to get easier interpretable solutions.

Running encoding and instance with the `dl` propagator results in the following six solutions corresponding to the solutions in Figure 6.[23] One for each possible permutation of tasks:

```
UNIX> python dl-app.py fsE.lp fsI.lp 0
clingo-dl version 1.0
Reading from fsE.lp ...
Solving...
Answer: 1
permutation(b,a) permutation(a,c) dl((a,1),1) dl((a,2),7)  \
dl((b,1),0) dl((b,2),1) dl((c,1),4) dl((c,2),11)
Answer: 2
permutation(b,a) permutation(c,b) dl((a,1),6) dl((a,2),16) \
dl((b,1),5) dl((b,2),10) dl((c,1),0) dl((c,2),5)
Answer: 3
permutation(c,b) permutation(a,c) dl((a,1),0) dl((a,2),3)  \
dl((b,1),8) dl((b,2),13) dl((c,1),3) dl((c,2),8)
Answer: 4
permutation(b,c) permutation(c,a) dl((a,1),6) dl((a,2),12) \
dl((b,1),0) dl((b,2),1) dl((c,1),1) dl((c,2),7)
Answer: 5
permutation(b,c) permutation(a,b) dl((a,1),0) dl((a,2),3)  \
```

[23] Note that in each solution all tasks are executed as early as possible. This is no coincidence and actually guaranteed by the algorithm implemented in the `Graph` class.

```
import sys
from clingo.application import Application, clingo_main
from clingo.ast import ProgramBuilder, parse_files
from dl import DLPropagator, HeadBodyTransformer, THEORY

class DLOptApp(Application):
    program_name = "clingo-dl-opt"
    version = "1.0"

    def __init__(self):
        self._bound = None
        self._propagator = DLPropagator()

    def _on_model(self, model):
        self._propagator.on_model(model)
        for symbol in model.symbols(theory=True):
            if symbol.match("dl", 2):
                n, v = symbol.arguments
                if n.match("bound", 0):
                    self._bound = v.number
                    break

    def main(self, ctl, files):
        ctl.register_propagator(self._propagator)
        ctl.add("base", [], THEORY)
        ctl.add("bound", ["b"], "&diff(head) { bound-0 } <= b.")

        with ProgramBuilder(ctl) as bld:
            hbt = HeadBodyTransformer()
            parse_files(files, lambda stm: bld.add(hbt.visit(stm)))

        ctl.ground([("base", [])])
        while ctl.solve(on_model=self._on_model).satisfiable:
            print("Found new bound: {}".format(self._bound))
            ctl.ground([("bound", [self._bound - 1])])

        if self._bound is not None: print("Optimum found")

sys.exit(int(clingo_main(DLOptApp(), sys.argv[1:])))
```

Listing 41. Application class `DLOptApp` for difference constraints with optimization (`dlO-app.py`)

```
dl((b,1),3) dl((b,2),7) dl((c,1),4) dl((c,2),13)
Answer: 6
permutation(c,a) permutation(a,b) dl((a,1),5) dl((a,2),10) \
dl((b,1),8) dl((b,2),14) dl((c,1),0) dl((c,2),5)
SATISFIABLE
```

7.2 Hybrid optimization with difference constraints

Finally, to find optimal solutions, we combine the algorithms in Listings 30 and 37 to minimize the total execution time of the tasks. The resulting algorithm is given in Listing 41.

As with the algorithm in Listing 37, a propagator and theory language is registered before solving and the program is parsed to accommodate a uniform semantic treatment. The control flow is similar to the branch-and-bound-based optimization algorithm in Listing 30 except that we now minimize the variable `bound`. More precisely, we minimize the difference between variable `0` and `bound` by adding the difference constraint $\texttt{0} - \texttt{bound} \leq b$ to the program in Line 26 where b is the best known execution time of the tasks as obtained from the assignment in Line 20 minus 1. To bound the maximum execution time of the tasks, we have to add one more line to the encoding in Listing 40:

```
18    &diff { (T,M)-bound } <= -D :- duration(T,M,D).
```

This makes sure that each task ends within the given bound. Running encoding and instance with the `dl` propagator results in the optimum bound 16 where the obtained solution corresponds to the lower left of the two optimal solutions indicated by a light blue background in Figure 6:

```
UNIX> python dl0-app.py fsE.lp fsI.lp
clingo-dl-opt version 1.0
Reading from fsE.lp ...
Solving...
Answer: 1
permutation(b,a) permutation(a,c) dl(bound,16) dl((a,1),1)   \
dl((a,2),7) dl((b,1),0) dl((b,2),1) dl((c,1),4) dl((c,2),11)
Found new bound: 16
Solving...
Optimum found
UNSATISFIABLE
```

8 Guess-and-check programming reloaded

Finally, we present an implementation of guess-and-check programming that relies on a combination of two *clingo* solvers. In contrast to the approach taken in Section 3.4, where the logic programs comprising the guess and check parts are combined in a single disjunctive program and thus solved by a single solver, the idea is now to deal with both programs separately by means of two interacting solvers. This last case-study nicely contrasts the efforts involved in meta-programming and the usage of solver APIs. Also, it further illustrates features of *clingo*'s API, namely, the manipulation of a program's abstract syntax tree (AST), the interaction of (multi-threaded) *clingo* instances via the propagator interface, the usage of assumptions during solving, and the addition of constraints to a program during runtime.

Unlike Section 3.4, we use `#program` directives to declare rules belonging to the guess and check programs. As above, guess atoms must not occur among the head atoms of the check program. For example, the simple guess and check programs from Listings 11 and 12 can now be rolled into one, as shown in Listing 42.

Passing this to our guess-and-check application `app.py` yields the same solution as with meta-programming:

```
UNIX> python app.py guess-check.lp 0
guess-and-check version 1.0
Reading from guess-check.lp
Solving...
```

```
1  #program guess.
2  1 { a(1..2) }.

4  #program check.
5  :- not a(1).
```

Listing 42. Guess-and-check program (`guess-check.lp`)

```
98  class GACApp(Application):
99      def __init__(self):
100         self.program_name = "guess-and-check"
101         self.version = "1.0"

103     def main(self, ctl, files):
104         check = []
105         with ast.ProgramBuilder(ctl) as builder:
106             trans = Transformer(builder, check)
107             ast.parse_files(files, trans.add)
108         ctl.register_propagator(GACPropagator(check))

110         ctl.ground([("base", [])])
111         ctl.solve()
```

Listing 43. The `GACApp` class for guess-and-check programming (`app.py`, Lines 98–111)

```
Answer: 1
a(2)
SATISFIABLE
```

In what follows, we detail the inner working of the approach. The idea is to have one solver guessing solution candidates, and another checking their compliance. Their interaction is realized through *clingo*'s propagator interface and restricted to testing total candidates, rather than partial ones as done in Section 7. In this way, the checking solver acts as a propagator within the guessing one.

The overall design is partitioned in four classes. Our description concentrates on these classes by following the overall workflow, although the line numbers reflect positions in the source code.

As before, we start from a derivative of *clingo*'s `Application` class and implement its `main` function as shown in Listing 43. As mentioned, the primary solver object `ctl` acts as the guesser, while the checker is encapsulated as its propagator. The `main` function starts by parsing the input programs, registers the propagator, grounds, and solves. The task of the `Transformer` in Line 106 is to add rules from the guess part to the program in the primary solver `ctl` via a `ProgramBuilder` and to collect rules from the check part in the list initialized in Line 104. This is done during parsing in Line 107 by means of the `add` function defined in the `Transformer` class.

The `add` function is given in Lines 14 to 27 of Listing 44 as the salient part of the `Transformer` class. It relies on variable `_state` to distinguish whether a rule is read in the context of a `guess` (or `base`) or `check` program. This variable is "toggled" in Lines 17 and 19 whenever a `#program` directive is encountered. Accordingly, the rule's AST is either added to the program builder of the guessing solver in Line 25 or appended to the

```
 8 class Transformer:
 9     def __init__(self, builder, check):
10         self._builder = builder
11         self._state = "guess"
12         self._check = check

14     def add(self, stm):
15         if stm.ast_type == ast.ASTType.Program:
16             if stm.name == "check" and not stm.parameters:
17                 self._state = "check"
18             elif stm.name in ("base", "guess") and not stm.parameters:
19                 self._state = "guess"
20             else:
21                 raise RuntimeError("unexpected program part")

23         else:
24             if self._state == "guess":
25                 self._builder.add(stm)
26             else:
27                 self._check.append(stm)
```

Listing 44. The `Transformer` class for classifying rules into the guess and check part (`app.py`, Lines 8–27)

`check` list (in Line 27) that is passed down from the `main` function to gather the check program.

Once parsing is finished, the filled list is used to initialize the propagator in Line 108. The corresponding `GACPropagator` class is given in Listing 45. It administers one or several solver objects, which are encapsulated by the `Checker` class in Listing 46. Given that no partial checks are performed, the propagation class only implements function `init` and `check` of *clingo*'s `Propagator` interface from Figure 3.

Let us first detail the initialization of the checkers in Lines 68 to 86. In fact, the `init` function may create several instances of the `Checker` class, depending on the number of threads of the primary solver. Each such checker (cf. Line 70) is initialized by looping over the atoms of the primary solver that provide the respective guess. While atoms are dropped in Line 76 that have been found to be false after grounding (and pre-processing) the guess program, either a fact or a choice rule is added to the checker in Lines 81 and 83 depending on whether the atom was found to be true or unknown, respectively. Clearly, unknown atoms of the guesser are most relevant to the checking solver, since their truth value is still subject to change. To this end, each checker comprises a dictionary mapping (unknown) guess literals to check literals; it is filled in Line 84.[24] Once all facts and choice rules are added to the checker, the check program gathered during parsing is grounded in Line 86 and added as well. The corresponding `add` and `ground` functions are defined in the `Checker` class and implemented in a straightforward way in the lines following Lines 37 and 40, respectively.

Just the same way as during initialization, both `GACPropagator` and `Checker` work

[24] In Section 3, this was done via predicate `guess/1`.

```
63 class GACPropagator(Propagator):
64     def __init__(self, check):
65         self._check = check
66         self._checkers = []

68     def init(self, init):
69         for _ in range(init.number_of_threads):
70             checker = Checker()
71             self._checkers.append(checker)

73             with checker.backend() as backend:
74                 for atom in init.symbolic_atoms:
75                     guess_lit = init.solver_literal(atom.literal)
76                     if init.assignment.is_false(guess_lit):
77                         continue

79                     check_lit = backend.add_atom(atom.symbol)
80                     if init.assignment.is_true(guess_lit):
81                         backend.add_rule([check_lit], [])
82                     else:
83                         backend.add_rule([check_lit], [], True)
84                         checker.add(guess_lit, check_lit)

86             checker.ground(self._check)

88     def check(self, control):
89         assignment = control.assignment
90         checker = self._checkers[control.thread_id]

92         if not checker.check(control):
93             conflict = []
94             for level in range(1, assignment.decision_level+1):
95                 conflict.append(-assignment.decision(level))
96             control.add_clause(conflict)
```

Listing 45. The `GACPropagator` class interfacing guessing and checking solver (`app.py`, Lines 63–96)

hand in hand in their respective `check` functions. The propagator's `check` function is called once the guessing solver has found a stable model; it immediately calls the `check` function of the associated checker in Line 92. In doing so, it passes along the `Control` object providing (limited) access to the underlying solver. This includes access to the assignment of the solver which of course corresponds to a model. The latter is at once extracted upon entering the checker's `check` function in Line 48 and analyzed afterwards. To this end, the function loops over the dictionary associating (originally unknown) guess and check atoms to transfer the guessed literals into a list of checker literals that are then used as assumptions in the subsequent call of the checker in Line 57. Technically, assumptions are added to the solver's assignment and amount semantically to the addition

```
29 class Checker:
30     def __init__(self):
31         self._ctl = Control()
32         self._map = []

34     def backend(self):
35         return self._ctl.backend()

37     def add(self, guess_lit, check_lit):
38         self._map.append((guess_lit, check_lit))

40     def ground(self, check):
41         with ast.ProgramBuilder(self._ctl) as builder:
42             for stm in check:
43                 builder.add(stm)

45         self._ctl.ground([("base", [])])

47     def check(self, control):
48         assignment = control.assignment

50         assumptions = []
51         for guess_lit, check_lit in self._map:
52             if assignment.is_true(guess_lit):
53                 assumptions.append(check_lit)
54             else:
55                 assumptions.append(-check_lit)

57         ret = self._ctl.solve(assumptions)
58         if ret.unsatisfiable is not None:
59             return ret.unsatisfiable

61         raise RuntimeError("search interrupted")
```

Listing 46. The `Checker` class wrapping the checking solver (`app.py`, Lines 29–61)

of integrity constraints (unlike externals;[25] cf. Section 5.1) In this way, the checker is forced to search for stable models comprising all guessed literals. If this fails, the checker's `check` succeeds, as does the propagator's `check`. Otherwise, the propagator extracts from the guesser's stable model all underlying decision literals and adds them as an integrity constraint, thus eliminating the combination of literals from the search space.

9 Discussion

This tutorial aims at enabling ASP users to become ASP engineers.

Our role model has been the landmark paper by Eén and Sörensson (2004) that aimed at *"give*[ing] *sufficient details about implementation to enable the reader to construct his or her own solver in a very short time. This will allow users of SAT-solvers to make*

[25] Also, assumptions only affect the current solve call. Opposed to this, assignments to externals persist over solve call (as long as the externals are not released, reassigned, or defined).

domain specific extensions or adaptions of current state-of-the-art SAT-techniques, to meet the needs of a particular application area." Their presentation of the C++ source code of the SAT solver *minisat* significantly boosted research in SAT by *"bridge*[ing] *the gap between existing descriptions of SAT-techniques and their actual implementation"* (Eén and Sörensson 2004).

We hope to achieve a similar effect with the tutorial at hand. However, unlike following suit in easing a white box approach to ASP solving, dealing with system modifications, we rather advocate the gray and black box approach put forward in the introduction, and make a case for application interface and meta programming, respectively. This is motivated by the much more elaborate model-ground-solve workflow of ASP systems that must often be addressed in its entirety to provide a certain functionality.

To this end, we describe several essential techniques for extending the ASP system *clingo* or implementing customized special-purpose systems. We have started with the lighter approach of meta programming in ASP and continued with application interface programming in Python (although several alternatives are available). Central to this is the new `Application` class of *clingo* that permits to draw on *clingo*'s infrastructure by starting processes similar to the one in *clingo*. This allows us to build customized ASP-based systems by overriding *clingo*'s `main` function, as illustrated by various examples throughout the tutorial. In particular, we have seen how derivatives of the `Application` class can be used to engage manipulations to programs' abstract syntax trees, control various forms of multi-shot solving, and set up theory propagators for foreign inferences. Multi-shot solving provides us with fine-grained control of ASP reasoning processes, while theory solving allows us to refine basic ASP solving by incorporating foreign types of constraints. Because of ASP's model-ground-solve methodology both techniques pervade its whole workflow, starting with extensions to the input language, over means for incremental and theory-enhanced grounding, to stateful and theory-enhanced solving. Multi-shot solving even adds a fourth dimension to *control* ASP reasoning processes.

Although meta programming has been around in ASP since its beginning (cf. Section 3 for a brief discussion), we hope that the reification feature of *clingo* makes it more attractive as a lightweight alternative to extend ASP systems. The idea of implementing ASP systems by pipelining was first advocated by Tomi Janhunen and used in his normalization toolbox (Janhunen and Niemelä 2011; Bomanson et al. 2014; Bomanson et al. 2016). In both cases, an intermediate ASP format is used to pass data from one solver to the next. While we used a fact-based representation of *aspif*, the normalization tools rely on the machine-oriented *smodels* format. Interestingly, *lc2casp* (Cabalar et al. 2016) implements a system for non-monotonic constraint solving by translating one *aspif* specification into another. That is, it takes the output of *gringo*, compiles out non-monotonicity, and feeds the result into *clingcon*, an extension of *clingo* with monotonic linear constraints over integers.

As mentioned in the introduction, *dlv* and *clingo* constitute nowadays the only genuine ASP systems in use. Accordingly, they are the only possible providers of native APIs for ASP. As detailed by Alviano et al. (2017), the latest versions of *dlv* combine the *idlv* grounder (Calimeri et al. 2017) with the ASP solver *wasp* (Alviano et al. 2015). As with *gringo*, the input language of *idlv* covers the second ASP language standard (Calimeri et al. 2019). Furthermore, it offers the integration of computable functions, similar to the mechanism (using terms preceded by '`@`') sketched at the beginning of Section 4. Unlike

this, a full-fledged Python API is offered by the ASP solver *wasp* (Dodaro and Ricca 2020). This has interesting applications to heuristic control and propagators for integrity constraints (Dodaro et al. 2016; Cuteri et al. 2020). In fact, former versions of *dlv* offer powerful Java integration (Febbraro et al. 2012), compliant with the Object-Relational Mapping standard (ORM), and implemented by wrapping *dlv* at its core. A Python library providing an ORM interface to *clingo* is also available (Rajaratnam 2021). A framework for developing applications embedding ASP on mobile devices is proposed by Fuscà et al. (2016).

Prior to the availability of APIs, various systems extending ASP have been built. For example, *dlvhex* (Redl 2016; Eiter et al. 2018) provides higher-order logic programs, whose higher-order atoms are implemented externally in C++ or Python; it is build upon *clingo*'s infrastructure. At the time, such a white box approach was only feasible thanks to a close collaboration between both groups at Vienna and Potsdam, not to mention that this fostered the development of *clingo*'s API quite a bit. Interestingly, also *clingcon* (Banbara et al. 2017), an extension of *clingo* with linear constraints over integers, started out as a white box approach and has just recently been transformed into a gray box approach, since otherwise its maintenance had been infeasible. Similarly yet much earlier, *adsolver* (Mellarkod et al. 2008) extended *smodels* (Niemelä and Simons 1997) with linear constraints over integers. Another category of ASP systems, such as *ezsmt* (Lierler and Susman 2016), *dingo* (Janhunen et al. 2011), and *aspmt* (Bartholomew and Lee 2014) translate ASP with constraints to SMT and use appropriate backends. Similarly, *mingo* (Liu et al. 2012) translates to Mixed Integer Linear Programming. Interestingly, the semantics of such hybrid ASP systems can be given in a theory-independent way by using denotational semantics (Cabalar et al. 2016; Cabalar et al. 2020b).

Both meta and application interface programming greatly facilitate the development of ASP-based systems and therefore ease the transposition of ideas into practice. The lighter approach of meta programming is well suited for rapid prototyping and moreover enjoys elaboration tolerance. However, once more control is needed, API programming is indispensable. Although it lacks full elaboration tolerance, it has nonetheless the great advantage to offer a high level of abstraction. This makes any project much easier to handle and to maintain than modifying the source code of an ASP system.

Last but not least, ASP has come a long way to turn into a mature and quite sophisticated approach to declarative problem solving. However, this sophistication should not become an obstacle to further technological advances. We hope that this tutorial contributes to coping with this challenge. After all, we, the ASP community, have the hard job of making our users' lives easy.

Acknowledgments

This work was partially funded by DFG grants SCHA 550/11 and 15.

We are grateful to the anonymous reviewers and in particular to Mirosław Truszczyński for his relentless efforts to improve the presentation of our paper — thank you so much!

Appendix A Saturation-based meta encoding

The saturation-based meta encoding in Listing 47 relies on a partition of the atoms of the input program induced by the strongly connect components of its positive dependency graph. Each atom and each loop of the program is contained in some part. The idea is to mimic the consecutive application of the immediate consequence operator to each component of the partition.

In a nutshell, the encoding in Listing 47 combines the following parts (Gebser et al. 2011):

1. guessing an interpretation (in Lines 14 to 33),
2. deriving the unsatisfiability-indicating atom `bot` if the interpretation is not a supported model (where each true atom occurs positively in the head of some rule whose body holds; cf. Lines 35 and 36),
3. deriving `bot` if the true atoms of some non-trivial strongly connected component are not acyclicly derivable (checked via determining the complement of a fixpoint of the immediate consequence operator; cf. Lines 38 to 63), and
4. saturating interpretations that do not correspond to stable models by deriving all truth assignments (for atoms) from `bot` (in Lines 65 and 66).

As an example, consider the simple logic program `a.lp`:

```
1 { a(1..2) }.
```

Computing its stable models with the meta encoding in Listing 47 (along with the auxiliary `#show` statements from Listing 48) yields the three expected models:

```
UNIX> clingo --output=reify --reify-sccs a.lp | \
      clingo - metaD.lp show.lp  0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
a(2)
Answer: 2
a(1)
Answer: 3
a(1) a(2)
SATISFIABLE
```

Now, the addition of an empty integrity constraint, namely "`:-.`", makes the program unsatisfiable. This is reflected by a single answer set containing all atoms of the program. This should not to be confused with the third model obtained above:

```
UNIX> clingo --output=reify --reify-sccs a.lp <(echo ":-.") | \
      clingo - metaD.lp show.lp  0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
a(1) a(2)
SATISFIABLE
```

In fact, additional show statements would reveal that the actual stable model also contains the artificial atom `bot` from which all atoms occurring in the original program are derivable (cf. Lines 65 and 66 in Listing 47). In other words, this special atom expresses the non-existence of stable models, and by saturating the model with all atoms it can only exist if no true stable models exist. This is because the semantics of disjunctive logic programs

```
sum(B,G,T) :- rule(_,sum(B,G)), T = #sum { W,L: weighted_literal_tuple(B,L,W) }.

supp(A,B) :- rule(     choice(H),B), atom_tuple(H,A).
supp(A,B) :- rule(disjunction(H),B), atom_tuple(H,A).

supp(A) :- supp(A,_).

atom(|L|) :- weighted_literal_tuple(_,L,_).
atom(|L|) :- literal_tuple(_,L).
atom( A ) :- atom_tuple(_,A).

fact(A) :- rule(disjunction(H),normal(B)), atom_tuple(H,A), not literal_tuple(B,_).

true(atom(A))               :- fact(A).
true(atom(A)); fail(atom(A)) :- supp(A), not fact(A).
               fail(atom(A)) :- atom(A), not supp(A).

true(normal(B)) :- literal_tuple(B),
  true(atom(L)): literal_tuple(B, L), L>0;
  fail(atom(L)): literal_tuple(B,-L), L>0.
fail(normal(B)) :- literal_tuple(B, L), fail(atom(L)), L>0.
fail(normal(B)) :- literal_tuple(B,-L), true(atom(L)), L>0.

true(sum(B,G)) :- sum(B,G,T),
  #sum {
    W,L: true(atom(L)), weighted_literal_tuple(B, L,W), L>0;
    W,L: fail(atom(L)), weighted_literal_tuple(B,-L,W), L>0
  } >= G.
fail(sum(B,G)) :- sum(B,G,T),
  #sum {
    W,L: fail(atom(L)), weighted_literal_tuple(B, L,W), L>0;
    W,L: true(atom(L)), weighted_literal_tuple(B,-L,W), L>0
  } >= T-G+1.

bot :- rule(disjunction(H),B), true(B), fail(atom(A)): atom_tuple(H,A).
bot :- true(atom(A)), fail(B): supp(A,B).

internal(C,normal(B)) :- scc(C,A), supp(A,normal(B)), scc(C,A'),
                         literal_tuple(B,A').
internal(C,sum(B,G))  :- scc(C,A), supp(A,sum(B,G)),  scc(C,A'),
                         weighted_literal_tuple(B,A',W).

external(C,normal(B)) :- scc(C,A), supp(A,normal(B)), not internal(C,normal(B)).
external(C,sum(B,G))  :- scc(C,A), supp(A,sum(B,G)),  not internal(C,sum(B,G)).

steps(C,Z-1) :- scc(C,_), Z = { scc(C,A): not fact(A) }.

wait(C,atom(A),0)   :- scc(C,A), fail(B): external(C,B), supp(A,B).
wait(C,normal(B),I) :- internal(C,normal(B)), steps(C,Z), I=0..Z-1,
                       fail(normal(B)).
wait(C,normal(B),I) :- internal(C,normal(B)), steps(C,Z), I<Z,
                       literal_tuple(B,A), wait(C,atom(A),I).

wait(C,sum(B,G),I)  :- internal(C,sum(B,G)), steps(C,Z), I=0..Z-1, sum(B,G,T),
  #sum {
    W,L:   fail(atom(L)),   weighted_literal_tuple(B, L,W), L>0, not scc(C,L);
    W,L: wait(C,atom(L),I), weighted_literal_tuple(B, L,W), L>0,     scc(C,L);
    W,L:   true(atom(L)),   weighted_literal_tuple(B,-L,W), L>0
  } >= T-G+1.
wait(C,atom(A),I)   :- wait(C,atom(A),0), steps(C,Z), I=1..Z,
                       wait(C,B,I-1): supp(A,B), internal(C,B).

bot :- scc(C,A), true(atom(A)), wait(C,atom(A),Z), steps(C,Z).

true(atom(A)) :- supp(A), not fact(A), bot.
fail(atom(A)) :- supp(A), not fact(A), bot.
```

Listing 47. A disjunctive meta encoding implementing saturation (`metaD.lp`)

```
1 #show.
2 #show X: output(X,B),     literal_tuple(B,A), true(atom(A)).
3 #show X: output(X,B), not literal_tuple(B,_).
```

Listing 48. Auxiliary `#show` statements for Listing 47 (`show.lp`)

is based on subset minimization. Saturation makes sure that `bot` is derived only if it is inevitable, that is, if it is impossible to construct any other models.[26]

Appendix B Intermediate language

To accommodate the rich input language, a general grounder-solver interface is needed. Although this could be left internal to *clingo*, it is good practice in ASP and neighboring fields to explicate such interfaces via an intermediate language. This also allows for using alternative downstream solvers or transformations.

Unlike the block-oriented *smodels* format, the *aspif* format is line-based. Notably, it abolishes the need of using symbol tables in *smodels*' format (Syrjänen 2001) for passing along meta-expressions and allows *gringo* to output information as soon as it is grounded. An *aspif* file starts with a header, beginning with the keyword `asp` along with version information and optional tags, viz.

$$\texttt{asp}\sqcup v_m \sqcup v_n \sqcup v_r \sqcup t_1 \sqcup \ldots \sqcup t_k$$

where v_m, v_n, v_r are non-negative integers representing the version in terms of *major*, *minor*, and *revision* numbers, and each t_i is a tag for $k \geq 0$. Currently, the only tag is `incremental`, meant to set up the underlying solver for multi-shot solving. An example header is given in the first lines of Listings 49 and 50 below. The rest of the file comprises one or more logic programs. Each logic program is a sequence of lines of *aspif* statements followed by a `0`, one statement or `0` per line, respectively. Positive and negative integers are used to represent positive or negative literals, respectively. Hence, `0` is not a valid literal.

Let us now briefly describe the format of *aspif* statements and illustrate them with the simple logic program in Listing 1 as well as the result of grounding a subset of Listing 33 only pertaining to difference constraints in Listing 50.

Rule statements have form

$$\texttt{1} \sqcup H \sqcup B$$

in which head H has form

$$h \sqcup m \sqcup a_1 \sqcup \ldots \sqcup a_m$$

where $h \in \{\texttt{0}, \texttt{1}\}$ determines whether the head is a disjunction or a choice, $m \geq 0$ is the number of head elements, and each a_i is an atom.

Body B has one of two forms:

[26] In fact, without the two saturating rules in Lines 65 and 66, Listing 47 would produce a stable model for each interpretation of the original program. The ones without `bot` represent stable models, while the ones with `bot` are mere interpretations. By saturation, all these interpretations are mapped to the set of all atoms. Given that the latter is a superset of all conceivable stable models, it can only exist if no stable models exist.

```
1  asp 1 0 0
2  1 1 1 1 0 0
3  1 0 1 2 0 1 1
4  1 0 1 3 0 1 -1
5  4 1 a 1 1
6  4 1 b 1 2
7  4 1 c 1 3
8  0
```

Listing 49. Representing the logic program from Listing 1 in *aspif* format

- Normal bodies have form

  $$\texttt{0}\sp n\sp l_1\sp \ldots\sp l_n$$

  where $n \geq 0$ is the length of the rule body, and each l_i is a literal.
- Weight bodies have form

  $$\texttt{1}\sp l\sp n\sp l_1\sp w_1\sp \ldots\sp l_n\sp w_n$$

  where l is a positive integer to denote the lower bound, $n \geq 0$ is the number of literals in the rule body, and each l_i and w_i are a literal and a positive integer.

All types of ASP rules are included in the above rule format. Heads are disjunctions or choices, including the special case of one-element disjunctions for representing normal rules. As in the *smodels* format, aggregate rules are restricted to one-element bodies, just that in *aspif* cardinality constraints are taken as special weight constraints. Otherwise, a body is simply a conjunction of literals.

The three rules in Listing 1 are represented by the statements in Lines 2–4 of Listing 49. For instance, the four occurrences of `1` in Line 2 capture a rule with a choice in the head, having one element, identified by `1`. The two remaining zeros capture a normal body with no element. For another example, Lines 2–7 of Listing 50 represent 6 of the facts in Listing 34, the four regular atoms in Lines 1–4 along two comprising theory atoms in Lines 11 and 12.

Minimize statements have form

$$\texttt{2}\sp p\sp n\sp l_1\sp w_1\sp \ldots\sp l_n\sp w_n$$

where p is an integer priority, $n \geq 0$ is the number of weighted literals, each l_i is a literal, and each w_i is an integer weight. Each of the above expressions gathers weighted literals sharing the same priority p from all `#minimize` directives and weak constraints in a logic program. As before, maximize statements are translated into minimize statements.

Projection statements result from `#project` directives and have form

$$\texttt{3}\sp n\sp a_1\sp \ldots\sp a_n$$

where $n \geq 0$ is the number of atoms, and each a_i is an atom.

Output statements result from `#show` directives and have form

$$\texttt{4}\sp m\sp s\sp n\sp l_1\sp \ldots\sp l_n$$

where $n \geq 0$ is the length of the condition, each l_i is a literal, and $m \geq 0$ is an integer

indicating the length in bytes of string s (where s excludes byte '\0' and newline). The output statements in Lines 5–7 of Listing 49 print the symbolic representation of atom `a`, `b`, or `c`, whenever the corresponding atom is true. For instance, the string '`a`' is printed if atom '`1`' holds. Unlike this, the statements in Lines 8–11 of Listing 50 unconditionally print the symbolic representation of the atoms stemming from the four facts in Lines 1–4 of Listing 34.

External statements result from `#external` directives and have form

$$5 \sqcup a \sqcup v$$

where a is an atom, and $v \in \{0, 1, 2, 3\}$ indicates free, true, false, and release.

Assumption statements have form

$$6 \sqcup n \sqcup l_1 \sqcup \ldots \sqcup l_n$$

where $n \geq 0$ is the number of literals, and each l_i is a literal. Assumptions instruct a solver to compute stable models containing $l_1, \ldots, l_n$. They are only valid for a single solver call.

Heuristic statements result from `#heuristic` directives and have form

$$7 \sqcup m \sqcup a \sqcup k \sqcup p \sqcup n \sqcup l_1 \sqcup \ldots \sqcup l_n$$

where $m \in \{0, \ldots, 5\}$ stands for the $(m{+}1)$th heuristic modifier among `level`, `sign`, `factor`, `init`, `true`, and `false`, a is an atom, k is an integer, p is a non-negative integer priority, $n \geq 0$ is the number of literals in the condition, and the literals l_i are the condition under which the heuristic modification should be applied.

Edge statements result from `#edge` directives and have form

$$8 \sqcup u \sqcup v \sqcup n \sqcup l_1 \sqcup \ldots \sqcup l_n$$

where u and v are integers representing an edge from node u to node v, $n \geq 0$ is the length of the condition, and the literals l_i are the condition for the edge to be present.

Let us now turn to the theory-specific part of *aspif*. Once a theory expression is grounded, *gringo* outputs a serial representation of its syntax tree. To illustrate this, we give in Listing 50 the (sorted) result of grounding all lines of Listing 33 related to difference constraints, viz. Lines 1–20 and Line 24.

```
 1  asp 1 0 0
 2  1 0 1 1 0 0
 3  1 0 1 2 0 0
 4  1 0 1 3 0 0
 5  1 0 1 4 0 0
 6  1 0 1 5 0 0
 7  1 0 1 6 0 0
 8  4 7 task(1) 0
 9  4 7 task(2) 0
10  4 15 duration(1,200) 0
11  4 15 duration(2,400) 0
12  9 0 1 200
13  9 0 3 400
14  9 0 6 1
```

```
15  9 0 11 2
16  9 1 0 4 diff
17  9 1 2 2 <=
18  9 1 4 1 -
19  9 1 5 3 end
20  9 1 8 5 start
21  9 2 7 5 1 6
22  9 2 9 8 1 6
23  9 2 10 4 2 7 9
24  9 2 12 5 1 11
25  9 2 13 8 1 11
26  9 2 14 4 2 12 13
27  9 4 0 1 10 0
28  9 4 1 1 14 0
29  9 6 5 0 1 0 2 1
30  9 6 6 0 1 1 2 3
31  0
```

Listing 50. *aspif* format (excerpt of result)

Theory terms are represented using the following statements:

$$\texttt{9}\sqcup\texttt{0}\sqcup u\sqcup w \tag{B1}$$

$$\texttt{9}\sqcup\texttt{1}\sqcup u\sqcup n\sqcup s \tag{B2}$$

$$\texttt{9}\sqcup\texttt{2}\sqcup u\sqcup t\sqcup n\sqcup u_1\sqcup\ldots\sqcup u_n \tag{B3}$$

where $n \geq 0$ is a length, index u is a non-negative integer, integer w represents a numeric term, string s of length n represents a symbolic term (including functions) or an operator, integer t is either `-1`, `-2`, or `-3` for tuple terms in parentheses, braces, or brackets, respectively, or an index of a symbolic term or operator, and each u_i is an integer for a theory term. Statements (B1), (B2), and (B3) capture numeric terms, symbolic terms, as well as compound terms (tuples, sets, lists, and terms over theory operators).

Fifteen theory terms are given in Lines 12–26 of Listing 50. Each of them is identified by a unique index in the third spot of each statement. While Lines 12–20 stand for primitive entities of type (B1) or (B2), the ones beginning with '`9␣2`' represent compound terms. For instance, Lines 21 and 22 represent `end(1)` or `start(1)`, respectively, and Line 23 corresponds to `end(1)-start(1)`.

Theory atoms are represented using the following statements:

$$\texttt{9}\sqcup\texttt{4}\sqcup v\sqcup n\sqcup u_1\sqcup\ldots\sqcup u_n\sqcup m\sqcup l_1\sqcup\ldots\sqcup l_m \tag{B4}$$

$$\texttt{9}\sqcup\texttt{5}\sqcup a\sqcup p\sqcup n\sqcup v_1\sqcup\ldots\sqcup v_n \tag{B5}$$

$$\texttt{9}\sqcup\texttt{6}\sqcup a\sqcup p\sqcup n\sqcup v_1\sqcup\ldots\sqcup v_n\sqcup g\sqcup u_1 \tag{B6}$$

where $n \geq 0$ and $m \geq 0$ are lengths, index v is a non-negative integer, a is an atom or `0` for directives, each u_i is an integer for a theory term, each l_i is an integer for a literal, integer p refers to a symbolic term, each v_i is an integer for a theory atom element, and integer g refers to a theory operator. Statement (B4) captures elements of theory atoms and directives, and statements (B5) and (B6) refer to the latter.

For instance, Line 27 captures the (single) theory element in '`{ end(1)-start(1) }`', and Line 29 represents the theory atom '`&diff { end(1)-start(1) } <= 200`'.

Comments have form

$$\texttt{10}\,␣\,s$$

where s is a string not containing a newline.

The *aspif* format constitutes the default output of *gringo* 5. With *clasp* 3.2, ground logic programs can be read in both *smodels* and *aspif* format. The tool *lpconvert* can be used to convert between both formats (Potassco Team 2021f).

References

Abels, D., Jordi, J., Ostrowski, M., Schaub, T., Toletti, A., and Wanko, P. 2021. Train scheduling with hybrid ASP. *Theory and Practice of Logic Programming 21,* 3, 317–347.

Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., and Zangari, J. 2017. The ASP system DLV2. In *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, M. Balduccini and T. Janhunen, Eds. Springer-Verlag, 215–221.

Alviano, M., Dodaro, C., Leone, N., and Ricca, F. 2015. Advances in WASP. See Calimeri et al. (2015), 40–54.

Balduccini, M., Lierler, Y., and Woltran, S., Eds. 2019. *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*. Springer-Verlag.

Banbara, M., Kaufmann, B., Ostrowski, M., and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming 17,* 4, 408–461.

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

Baral, C. and Gelfond, M. 2000. Reasoning agents in dynamic domains. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers, Dordrecht, 257–279.

Barrett, C., Sebastiani, R., Seshia, S., and Tinelli, C. 2009. Satisfiability modulo theories. See Biere et al. (2009), Chapter 26, 825–885.

Bartholomew, M. and Lee, J. 2014. System aspmt2smt: Computing ASPMT theories by SMT solvers. See Fermé and Leite (2014), 529–542.

Biere, A., Heule, M., van Maaren, H., and Walsh, T., Eds. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

Bomanson, J., Gebser, M., and Janhunen, T. 2014. Improving the normalization of weight rules in answer set programs. See Fermé and Leite (2014), 166–180.

Bomanson, J., Gebser, M., and Janhunen, T. 2016. Rewriting optimization statements in answer-set programs. See Carro and King (2016), 5:1–5:15.

Brewka, G., Delgrande, J., Romero, J., and Schaub, T. 2015. Implementing preferences with asprin. See Calimeri et al. (2015), 158–172.

Brewka, G., Eiter, T., and McIlraith, S., Eds. 2012. *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*. AAAI Press.

Cabalar, P., Fandinno, J., Garea, J., Romero, J., and Schaub, T. 2020. eclingo: A solver for epistemic logic programs. *Theory and Practice of Logic Programming 20,* 5, 834–847. `https://github.com/potassco/eclingo`.

Cabalar, P., Fandinno, J., Schaub, T., and Wanko, P. 2020a. An ASP semantics for constraints involving conditional aggregates. In *Proceedings of the Twenty-fourth European Conference on Artificial Intelligence (ECAI'20)*, G. De Giacomo, A. Catalá, B. Dilkina,

M. Milano, S. Barro, A. Bugarín, and J. Lang, Eds. Frontiers in Artificial Intelligence and Applications, vol. 325. IOS Press, 664–671.

Cabalar, P., Fandinno, J., Schaub, T., and Wanko, P. 2020b. A uniform treatment of aggregates and constraints in hybrid ASP. In *Proceedings of the Seventeenth International Conference on Principles of Knowledge Representation and Reasoning (KR'18)*, D. Calvanese, E. Erdem, and M. Thielscher, Eds. AAAI Press, 193–202.

Cabalar, P., Kaminski, R., Morkisch, P., and Schaub, T. 2019. telingo = ASP + time. See Balduccini et al. (2019), 256–269.

Cabalar, P., Kaminski, R., Ostrowski, M., and Schaub, T. 2016. An ASP semantics for default reasoning with constraints. In *Proceedings of the Twenty-fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, R. Kambhampati, Ed. IJCAI/AAAI Press, 1015–1021.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., and Schaub, T. 2019. ASP-Core-2 input language format. *Theory and Practice of Logic Programming 20,* 2, 294–309.

Calimeri, F., Fuscà, D., Perri, S., and Zangari, J. 2017. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale 11,* 1, 5–20.

Calimeri, F., Ianni, G., and Truszczyński, M., Eds. 2015. *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*. Springer-Verlag.

Carro, M. and King, A., Eds. 2016. *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16)*. OpenAccess Series in Informatics (OASIcs), vol. 52. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

CFFI 2021. Cffi documentation. `https://cffi.readthedocs.io`.

Cotton, S. and Maler, O. 2006. Fast and flexible difference constraint propagation for DPLL(T). In *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, A. Biere and C. Gomes, Eds. Springer-Verlag, 170–183.

Cuteri, B., Dodaro, C., Ricca, F., and Schüller, P. 2020. Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI'20)*, C. Bessiere, Ed. ijcai.org, 1688–1694.

Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys 33,* 3, 374–425.

Delgrande, J., Schaub, T., and Tompits, H. 2003. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming 3,* 2, 129–187.

Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J., and Schaub, T. 2018. plasp 3: Towards effective ASP planning. *Theory and Practice of Logic Programming 19,* 3, 477–504.

Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F., and Schekotihin, K. 2016. Combining answer set programming and domain heuristics for solving hard industrial problems. *Theory and Practice of Logic Programming 16,* 5-6, 653–669.

Dodaro, C. and Ricca, F. 2020. The external interface for extending WASP. *Theory and Practice of Logic Programming 20,* 2, 225–248.

Eén, N. and Sörensson, N. 2004. An extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, E. Giunchiglia and A. Tacchella, Eds. Springer-Verlag, 502–518.

Eiter, T., Erdem, E., Erdogan, H., and Fink, M. 2013. Finding similar/diverse solutions in answer set programming. *Theory and Practice of Logic Programming 13,* 3, 303–359.

Eiter, T., Faber, W., Leone, N., and Pfeifer, G. 2003. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory and Practice of Logic Programming 3,* 4-5, 463–498.

Eiter, T., Germano, S., Ianni, G., Kaminski, T., Redl, C., Schüller, P., and Weinzierl, A. 2018. The DLVHEX system. *Künstliche Intelligenz 32,* 2-3, 187–189.

Eiter, T. and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence 15,* 3-4, 289–323.

Eiter, T. and Polleres, A. 2006. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming 6,* 1-2, 23–60.

Febbraro, O., Leone, N., Grasso, G., and Ricca, F. 2012. JASP: A framework for integrating answer set programming with Java. See Brewka et al. (2012), 541–551.

Fermé, E. and Leite, J., Eds. 2014. *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*. Springer-Verlag.

Frioux, C., T.Schaub, Schellhorn, S., Siegel, A., and Wanko, P. 2019. Hybrid metabolic network completion. *Theory and Practice of Logic Programming 19,* 1, 83–108.

Fuscà, D., Germano, S., Zangari, J., Anastasio, M., Calimeri, F., and Perri, S. 2016. A framework for easing the development of applications embedding answer set programming. In *Proceedings of the Eighteenth International Symposium on Principles and Practice of Declarative Programming (PPDP'16)*, J. Cheney and G. Vidal, Eds. ACM Press, 38–49.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., and Schaub, T. 2015. Abstract Gringo. *Theory and Practice of Logic Programming 15,* 4-5, 449–463.

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., and Thiele, S. 2015. *Potassco User Guide*, 2 ed. University of Potsdam.

Gebser, M., Kaminski, R., Kaufmann, B., Lühne, P., Obermeier, P., Ostrowski, M., Romero, J., Schaub, T., Schellhorn, S., and Wanko, P. 2018. The Potsdam answer set solving collection 5.0. *Künstliche Intelligenz 32,* 2-3, 181–182.

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

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. 2008. Engineering an incremental ASP solver. In *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, M. Garcia de la Banda and E. Pontelli, Eds. Springer-Verlag, 190–205.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P. 2016. Theory solving made easy with clingo 5. See Carro and King (2016), 2:1–2:15.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2012. *Answer Set Solving in Practice.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming 19,* 1, 27–82.

Gebser, M., Kaminski, R., and Schaub, T. 2011. Complex optimization in answer set programming. *Theory and Practice of Logic Programming 11,* 4-5, 821–839.

Gebser, M., Kaufmann, B., and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187-188*, 52–89.

Gebser, M., Pührer, J., Schaub, T., and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, D. Fox and C. Gomes, Eds. AAAI Press, 448–453.

Gebser, M. and Schaub, T. 2016. Modeling and language extensions. *AI Magazine 37,* 3, 33–44.

Gelfond, M. and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press.

Gelfond, M. and Lifschitz, V. 1990. Logic programs with classical negation. In *Proceedings*

of the Seventh International Conference on Logic Programming (ICLP'90), D. Warren and P. Szeredi, Eds. MIT Press, 579–597.

Gelfond, M. and Son, T. 1997. Reasoning with prioritized defaults. In *Third International Workshop on Logic Programming and Knowledge Representation*, J. Dix, L. Pereira, and T. Przymusinski, Eds. Springer-Verlag, 164–223.

Gelfond, M. and Zhang, Y. 2019. Vicious circle principle, aggregates, and formation of sets in ASP based languages. *Artificial Intelligence 275*, 28–77.

Heyting, A. 1930. Die formalen Regeln der intuitionistischen Logik. In *Sitzungsberichte der Preussischen Akademie der Wissenschaften*. Deutsche Akademie der Wissenschaften zu Berlin, 42–56.

Janhunen, T., Kaminski, R., Ostrowski, M., Schaub, T., Schellhorn, S., and Wanko, P. 2017. Clingo goes linear constraints over reals and integers. *Theory and Practice of Logic Programming 17,* 5-6, 872–888.

Janhunen, T., Liu, G., and Niemelä, I. 2011. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Proceedings of the First Workshop on Grounding and Transformation for Theories with Variables (GTTV'11)*, P. Cabalar, D. Mitchell, D. Pearce, and E. Ternovska, Eds. 1–13.

Janhunen, T. and Niemelä, I. 2011. Compact translations of non-disjunctive answer set programs to propositional clauses. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of his 65th Birthday*, M. Balduccini and T. Son, Eds. Springer-Verlag, 111–130.

Kaminski, R., Schaub, T., and Wanko, P. 2017. A tutorial on hybrid answer set solving with clingo. In *Proceedings of the Thirteenth International Summer School of the Reasoning Web*, G. Ianni, D. Lembo, L. Bertossi, W. Faber, B. Glimm, G. Gottlob, and S. Staab, Eds. Springer-Verlag, 167–203.

Kaufmann, B., Leone, N., Perri, S., and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Magazine 37,* 3, 25–32.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

Lierler, Y. and Susman, B. 2016. SMT-based constraint answer set solver EZSMT. See Carro and King (2016), 1:1–1:15.

Lifschitz, V. 2019. *Answer Set Programming.* Springer-Verlag.

Lifschitz, V., Pearce, D., and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic 2,* 4, 526–541.

Liu, G., Janhunen, T., and Niemelä, I. 2012. Answer set programming via mixed integer programming. See Brewka et al. (2012), 32–42.

Marques-Silva, J., Lynce, I., and Malik, S. 2009. Conflict-driven clause learning SAT solvers. See Biere et al. (2009), Chapter 4, 131–153.

McCarthy, J. 1998. Elaboration tolerance.

Mellarkod, V., Gelfond, M., and Zhang, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 251–287.

Niemelä, I. and Simons, P. 1997. Smodels: An implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, J. Dix, U. Furbach, and A. Nerode, Eds. Springer-Verlag, 420–429.

Nieuwenhuis, R., Oliveras, A., and Tinelli, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM 53,* 6, 937–977.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., and Barry, M. 2001. An A-prolog decision support system for the space shuttle. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, I. Ramakrishnan, Ed. Springer-Verlag, 169–183.

Oikarinen, E. and Janhunen, T. 2006. Modular equivalence for normal logic programs. In *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, Eds. IOS Press, 412–416.

Pearce, D. 1997. A new logical characterisation of stable models and answer sets. In *Proceedings of the Sixth International Workshop on Non-Monotonic Extensions of Logic Programming (NMELP'96)*, J. Dix, L. Pereira, and T. Przymusinski, Eds. Springer-Verlag, 57–70.

Potassco Team. 2021a. Answer set solving in practice, course material. `https://teaching.potassco.org`.

Potassco Team. 2021b. *clingo*. `https://github.com/potassco/clingo`.

Potassco Team. 2021c. *clingo*'s application programming interface. `https://potassco.org/clingo/python-api/5.5`.

Potassco Team. 2021d. Difference constraints propagator for *clingo* in Python. `https://github.com/potassco/clingo/tree/master/examples/clingo/dl`.

Potassco Team. 2021e. Guess-and-check programming in Python. `https://github.com/potassco/clingo/tree/master/examples/reify/gac`.

Potassco Team. 2021f. *lpconvert*. `https://github.com/potassco/libpotassco`.

Potassco Team. 2021g. Meta encodings. `https://github.com/potassco/clingo/tree/master/examples/reify`.

Potassco Team. 2021h. Solving the towers of hanoi problem with *clingo*. `https://github.com/potassco/clingo/tree/master/examples/clingo/opt`.

Rajaratnam, D. 2021. `https://github.com/potassco/clorm`.

Redl, C. 2016. The dlvhex system for knowledge representation: recent advances. *Theory and Practice of Logic Programming 16,* 5-6, 866–883.

Romero, J., Schaub, T., and Wanko, P. 2016. Computing diverse optimal stable models. See Carro and King (2016), 3:1–3:14.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

Son, T., Baral, C., Nam, T., and McIlraith, S. 2006. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic 7,* 4, 613–657.

Syrjänen, T. 2001. Lparse 1.0 user's manual.

Wikipedia contributors. 2021. Metaprogramming — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=1001427050`.