Theory Solving Made Easy with *Clingo* 5*

Martin Gebser[1], Roland Kaminski[1], Benjamin Kaufmann[1], Max Ostrowski[1], Torsten Schaub[1,2], and Philipp Wanko[1]

1 **University of Potsdam, Germany**
2 **INRIA Rennes, France**

Abstract

Answer Set Programming (ASP) is a model, ground, and solve paradigm. The integration of application- or theory-specific reasoning into ASP systems thus impacts on many if not all elements of its workflow, viz. input language, grounding, intermediate language, solving, and output format. We address this challenge with the fifth generation of the ASP system *clingo* and its grounding and solving components by equipping them with well-defined generic interfaces facilitating the manifold integration efforts. On the grounder's side, we introduce a generic way of specifying language extensions and propose an intermediate format accommodating their ground representation. At the solver end, this is accompanied by high-level interfaces easing the integration of theory propagators dealing with these extensions.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Answer Set Programming, Theory Language, Theory Propagation

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.NNN

1 Introduction

The *clingo* system, along with its grounding and solving components *gringo* and *clasp*, is nowadays among the most widely used tools for Answer Set Programming (ASP; [20]). This does not only apply to end-users, but more and more to system developers who build upon *clingo*'s infrastructure for developing their own systems. Among them, we find (alphabetically) *clasp-nk* [12], *clingcon* [23], *dflat* [1], *dingo* [19], *dlvhex* [13], *inca* [11], and *mingo* [21]. None of these systems can use *clingo* or its components without workarounds or even involved modifications to realize the desired functionality. Moreover, since ASP is a model, ground, and solve paradigm, such modifications are rarely limited to a single component but often spread throughout the whole workflow. This begins with the addition of new language constructs to the input language, requiring in turn amendments to the grounder as well as syntactic means for passing the ground constructs to a downstream system. In case they are to be dealt with by an ASP solver, it must be enabled to treat the specific input and incorporate corresponding solving capacities. Finally, each such extension is application-specific and requires different means at all ends.

We address this challenge with the new *clingo* series 5 and its components. This is accomplished by introducing generic interfaces that allow for accommodating extensions to ASP at the salient stages of its workflow. To begin with, we extend *clingo*'s grounder component *gringo* with means for specifying simple theory grammars in which new theories can be represented. As theories are expressed using constructs close to ASP's basic modeling language, the existing grounding machinery takes care of instantiating them. This also

* This work was partially supported by DFG-SCHA-550/9

Technical Communications of the 32nd Int'l Conference on Logic Programming (ICLP 2016).
Editors: Manuel Carro, Andy King, Marina De Vos, and Neda Saeedloei; Article No. NNN; pp. NNN:1–NNN:14

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

involves a new intermediate ASP format that allows for passing the enriched information from grounders to solvers in a transparent way. (Since this format is mainly for settings with stand-alone grounders and solvers, and thus outside the scope of *clingo*, we delegate details to [15].) For a complement, *clingo* 5 provides several interfaces for reasoning with theory expressions. On the one hand, the existing Lua and Python APIs are extended by high-level interfaces for augmenting propagation in *clasp* with so-called *theory propagators*. Several such propagators can be registered with *clingo*, each implementing an interface of four basic methods. Our design is driven by the objective to provide means for rapid prototyping of dedicated reasoning procedures while enabling effective implementations. To this end, the interface supports, for instance, stateful theory propagators as well as multi-threading in the underlying solver. On the other hand, the functionality of the aforementioned extended APIs is now also offered via a C interface. This is motivated by the wide availability of foreign function interfaces for C, which enable the import of *clingo* in programming languages like Java or Haskell. A first application of this is the integration of *clingo* 5 into SWI-Prolog.[1]

2 Input Language

This section introduces the novel features of *clingo* 5's input language. All of them are situated in the underlying grounder *gringo* 5 and can thus also be used independently of *clingo*. We start with a detailed description of *gringo* 5's generic means for defining theories and afterwards summarize further new features.

Our 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 1, 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 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 $1 \leq i \leq n$.[2] Note that solving difference constraints is polynomial, while solving linear equations (over integers) is NP-hard. The theory language for expressing both types of constraints is defined in Lines 1–13 and preceded by the directive `#theory`. The elements of the resulting theory language are preceded by `&` and used as regular atoms in the logic program in Lines 15–21.

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

```
#theory T {D1;...;Dn}.
```

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 form

```
&p/k : t,o      or      &p/k : t,{◇1,...,◇m},t',o
```

where p is a predicate name and k its arity, t, 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 9–12 of Listing 1. The language of a theory atom definition as above contains all *theory atoms* of form

[1] `https://github.com/JanWielemaker/clingo`

[2] For simplicity, we consider normalized difference constraints rather than general ones of form $x_1 - x_2 \circ k$.

```
1  #theory difference {
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; - : 0, binary, left };
7    domain_term { .. : 1, binary, left };
8    show_term   { / : 1, binary, left };
9    &dom/0 : domain_term, {=}, linear_term, any;
10   &sum/0 : linear_term, {<=,=,>=,<,>,!=}, linear_term, any;
11   &diff/0 : diff_term, {<=}, constant, any;
12   &show/0 : show_term, directive
13 }.

15 #const n=2.  #const m=1000.
16 task(1..n).  duration(T,200*T) :- task(T).
17 &dom  { 1..m } = start(T) :- task(T).
18 &dom  { 1..m } = end(T)   :- task(T).
19 &diff { end(T)-start(T) } <= D :- duration(T,D).
20 &sum  { end(T) : task(T); -start(T) : task(T) } <= m.
21 &show { start/1; end/1 }.
```

Listing 1 Logic program enhanced with difference and linear constraints (`diff.lp`)

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

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. Five occurrences of theory atoms can be found in Lines 17–21 of Listing 1.

A *theory term definition* has 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 respective definition specifies the language of all theory terms that can be constructed via its operators. Examples of theory term definitions are given in Lines 2–8 of Listing 1. 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 form

$$\diamond\ \texttt{:}\ p\texttt{,unary} \quad \text{or} \quad \diamond\ \texttt{:}\ p\texttt{,binary,}a$$

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 Line 5 of Listing 1, where the `binary` operator `*`

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

4  &dom  { 1..1000 } = start(1).          &dom  { 1..1000 } = start(2).
5  &dom  { 1..1000 } = end(1).            &dom  { 1..1000 } = end(2).
6  &diff { end(1)-start(1) } <= 200.      &diff { end(2)-start(2) } <= 400.
7  &sum  { end(1); end(2); -start(1);     -start(2) } <= 1000.
8  &show { start/1; end/1 }.
```

Listing 2 Human-readable result of grounding Listing 1 via '`gringo -text diff.lp`'

is defined with precedence `1` and `left` associativity. In total, Lines 2–8 of Listing 1 include nine theory operator definitions. Particular *theory operators* can be assembled (written consecutively without spaces) from the symbols '`!`', '`<`', '`=`', '`>`', '`+`', '`-`', '`*`', '`/`', '`\`', '`?`', '`&`', '`|`', '`.`', '`:`', '`;`', '`~`', and '`^`'. For instance, in Line 7 of Listing 1, 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, standard grounding techniques can be used for gathering theory terms. (However, the actual atom within a theory atom comprises regular terms only.) 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 nicely seen on the different results in Listing 2 of grounding terms formed with the regular and theory-specific variants of operator '`..`'. Observe that the fact `task(1..n)` in Line 16 of Listing 1 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 16 of Listing 1 to the theory term '`end(T)-start(T)`' in Line 19. While each instance of '`200*T`' is evaluated during grounding, instances of the theory term are left in Line 6 of Listing 2. 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`'.

The remainder of this section is dedicated to other language extensions of *gringo* 5 aiming at a disentanglement of the various uses of `#show` directives (and their induced symbol table). Such directives were beforehand used for controlling the output of stable models, delineating the scope of reasoning modes (e.g., intersection, union, projection, etc.), and for passing special-purpose information to downstream systems. For instance, theory and heuristic information was passed to *clasp* via dedicated predicates like `_edge` and `_heuristic`. This entanglement brought about several shortcomings. In fact, passing information via a symbol table did not only scramble the output, but also provoked overhead in grounding and filtering "artificial" symbolic information.

Now, in *gringo* 5, the sole purpose of `#show` is to furnish an output directive. There are three different kinds of such statements:

```
#show.      #show p/n.      #show t : l_1,...,l_n.
```

The first form hides all atoms, and the second all except those over predicates p/n indicated by `#show` statements. The third form does not hide any atoms and can be used to output arbitrary terms t, whenever the literals $l_1, \ldots, l_n$ in the condition after '`:`' hold. This is

particularly useful in meta-programming, e.g., '`#show A : holds(A).`' can be used to map back reified atoms.

Atoms used in reasoning modes are indicated by `#project` directives, having two forms:

```
#project p/n.    #project a : l1,...,ln.
```

Here, p is a predicate name with arity n, a is an atom, and $l_1, \ldots, l_n$ are literals. While the first form declares all atoms over predicate p/n as subject to projection, the second includes instances of a obtained via grounding, as detailed in [16] for `#external` directives.

The last two new directives of interest abolish the need for the special-purpose predicates `_edge` and `_heuristic`, previously used in conjunction with the ASP solver *clasp*:

```
#edge (u,v) : l1,...,ln.
#heuristic a : l1,...,ln. [k@p,m]
```

As above, a is an atom, and $l_1, \ldots, l_n$ are literals. Moreover, u, v, k, p, m are terms, where '`(u,v)`' stands for an edge from u to v in an acyclicity extension [7]. Integer values for k and p along with `init`, `factor`, `level`, `sign`, `true`, or `false` for m determine a heuristic modifier [17]. Finally, note that zero is taken as default priority when the optional '`@`p' part in '`[`k`@`p`,`m`]`', resembling the syntax of ranks for weak constraints [9], is omitted.

3 Logical Characterization

The semantics of logic programs modulo theories rests upon ground programs P over two disjoint alphabets, $\mathcal{A}$ and $\mathcal{T}$, consisting of regular and *theory atoms*. Accordingly, P is a set of rules r of the form $h \leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$, where the head h is constant $\bot$, a_0 or $\{a_0\}$ for an atom $a_0 \in \mathcal{A} \cup \mathcal{T}$, and $\{a_1, \ldots, a_n\} \subseteq \mathcal{A} \cup \mathcal{T}$. If $h = \bot$, r is called an *integrity constraint*, a *normal rule* if $h = a_0$, or a *choice rule* if $h = \{a_0\}$; as usual, we skip $\bot$ when writing integrity constraints. We let $h(r) = \emptyset$ for an integrity constraint r, $h(r) = \{a_0\}$ for a normal or choice rule r, and define $h(P) = \bigcup_{r \in P} h(r)$ as the *head atoms* of P. In analogy to inputs atoms from `#external` directives [16], we partition $\mathcal{T}$ into *defined* theory atoms $\mathcal{T} \cap h(P)$ and *external* theory atoms $\mathcal{T} \setminus h(P)$.

Given a collection $\mathbb{T}$ of theories, we associate each $T \in \mathbb{T}$ with a *scope* $\mathcal{T}^T$ of atoms relevant to T, and let $\mathcal{T} = \bigcup_{T \in \mathbb{T}} \mathcal{T}^T$ be the corresponding set of theory atoms. Reconsidering the input language in Section 2, a natural choice for $\mathcal{T}^T$ consists of all (ground) atoms declared within a `#theory` directive for T. However, as we see in Section 5, a scope may in general include atoms written in regular as well as extended syntax (the latter preceded by '`&`') in the input language.

In order to reflect different forms of theory propagation, we further consider a partition of the scope $\mathcal{T}^T$ of a theory T into *strict* theory atoms $\mathcal{T}_e^T$ and *non-strict* theory atoms $\mathcal{T}_i^T$ such that $\mathcal{T}_e^T \cap \mathcal{T}_i^T = \emptyset$ and $\mathcal{T}_e^T \cup \mathcal{T}_i^T = \mathcal{T}^T$. The strict theory atoms in $\mathcal{T}_e^T$ resemble equivalences as expressed by the constraint atoms of *clingcon* [23], which must be assigned to true iff their associated constraints hold. This is complemented by viewing the non-strict theory atoms in $\mathcal{T}_i^T$ as implications similar to the constraint statements of *ezcsp* [2], where only statements assigned to true impose requirements, while constraints associated with false ones are free to hold or not. Given the distinction of respective kinds of theory atoms, a combined theory T may integrate constraints according to the semantics of *clingcon* and *ezcsp*, e.g., indicated by dedicated predicates or arguments thereof in T's theory language.

We now turn to mapping the semantics of logic programs modulo theories back to regular stable models. In the abstract sense, we call any $\mathcal{S}^T \subseteq \mathcal{T}^T$ a *T-solution* if T is consistent with the conditions expressed by elements of $\mathcal{S}^T$ as well as the complements of conditions

associated with the false strict theory atoms in $\mathcal{T}_e^T \setminus \mathcal{S}^T$.[3] Generalizing this concept to a collection $\mathbb{T}$ of theories, we say that $\mathcal{S} \subseteq \mathcal{T}$ is a $\mathbb{T}$-*solution* if $\mathcal{S} \cap \mathcal{T}^T$ is a T-solution for each $T \in \mathbb{T}$. Then, we define a set $X \subseteq \mathcal{A} \cup \mathcal{T}$ of (regular and theory) atoms as a $\mathbb{T}$-*stable model* of a ground program P if there is some $\mathbb{T}$-solution $\mathcal{S}$ such that X is a (regular) stable model of the program

$$P \cup \{a \leftarrow \mid T \in \mathbb{T}, a \in (\mathcal{T}_e^T \setminus h(P)) \cap \mathcal{S}\} \cup \{\leftarrow \sim a \mid T \in \mathbb{T}, a \in (\mathcal{T}_e^T \cap h(P)) \cap \mathcal{S}\} \quad (1)$$

$$\cup \{\{a\} \leftarrow \mid T \in \mathbb{T}, a \in (\mathcal{T}_i^T \setminus h(P)) \cap \mathcal{S}\} \cup \{\leftarrow a \mid T \in \mathbb{T}, a \in (\mathcal{T}^T \cap h(P)) \setminus \mathcal{S}\}. \quad (2)$$

That is, the rules added to P in (1) and (2) express conditions aligning $X \cap \mathcal{T}$ with an underlying $\mathbb{T}$-solution $\mathcal{S}$. First, the facts in (1) make sure that external theory atoms that are strict, i.e., included in $\mathcal{T}_e^T \setminus h(P)$ for some $T \in \mathbb{T}$, and hold in $\mathcal{S}$ belong to X as well. Unlike this, the corresponding set of choice rules in (2) merely says that non-strict external theory atoms from $\mathcal{S}$ may be included in X, thus not insisting on a perfect match between non-strict theory atoms and elements of $\mathcal{S}$. Moreover, the integrity constraints in (1) and (2) take care of defined theory atoms belonging to $h(P)$. The respective set in (1) again focuses on strict theory atoms and stipulates the ones from $\mathcal{S}$ to be included in X as well. In addition, for both strict and non-strict defined theory atoms, the integrity constraints in (2) assert the falsity of atoms that do not hold in $\mathcal{S}$.

For example, consider a program $P = \{a \leftarrow b, \sim c\}$ subject to some theory T with the strict and non-strict theory atoms $\mathcal{T}_e^T = \{a, b\}$ and $\mathcal{T}_i^T = \{c\}$, and let $\mathcal{S} = \{a, b, c\}$ be a T-solution. Then, the extended program for $\mathcal{S}$ is $P \cup \{b \leftarrow ; \{c\} \leftarrow ; \leftarrow \sim a\}$, whose (only) regular stable model $X = \{a, b\}$ is a $\{T\}$-stable model of P. Note that $\mathcal{S}$ assigns the non-strict theory atom c to true, while X excludes it to keep $a \leftarrow b, \sim c$ applicable for the (strict) defined theory atom a.

To summarize the main principles of the $\mathbb{T}$-stable model concept, strict theory atoms (for some $T \in \mathbb{T}$) must exactly match their interpretation in a $\mathbb{T}$-solution $\mathcal{S}$, while non-strict ones (not strict for any $T \in \mathbb{T}$) in X are only required not to exceed $\mathcal{S}$. Second, external theory atoms that hold in $\mathcal{S}$ are mapped to facts or choice rules, while conditions on defined ones are enforced by means of integrity constraints. As a result, $\mathbb{T}$-stable models are understood as regular stable models, yet relative to extensions of a given program P determined by underlying $\mathbb{T}$-solutions. Notably, the concept of $\mathbb{T}$-stable models also carries on to logic programs allowing for further constructs, such as weight constraints and disjunction, which have not been discussed here for brevity (cf. [24]).

4 Algorithmic Characterization

As detailed in [18], a ground program P induces *completion* and *loop nogoods*, given by $\Delta_P = \Delta_{B(P)} \cup \Delta_{\mathcal{A} \cup (\mathcal{T} \cap h(P))}$ or $\Lambda_P = \bigcup_{\emptyset \subset U \subseteq \mathcal{A} \cup (\mathcal{T} \cap h(P))} \{\lambda(a, U) \mid a \in U\}$, respectively, where $B(P)$ is the set of rule bodies occurring in P. Note that both sets of nogoods are restricted by regular atoms and defined theory atoms, while external theory atoms can a priori be assigned freely, although any occurrences in rule bodies are subject to evaluation via respective nogoods in $\Delta_{B(P)}$. A (partial) *assignment* $\mathbf{A}$ is a consistent set of *(signed) literals* of the form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in (\mathcal{A} \cup \mathcal{T}) \cup B(P)$, i.e., $\{\mathbf{T}v, \mathbf{F}v\} \not\subseteq \mathbf{A}$ for all $v \in (\mathcal{A} \cup \mathcal{T}) \cup B(P)$; $\mathbf{A}$ is *total* if $\{\mathbf{T}v, \mathbf{F}v\} \cap \mathbf{A} \neq \emptyset$ for all $v \in (\mathcal{A} \cup \mathcal{T}) \cup B(P)$. We say that some nogood δ is

[3] Although we omit formal details, atoms in Satisfiability Modulo Theories (SMT; [4]) belong to first-order predicates interpreted in a theory T, and the ones that hold in some model of T provide a T-solution $\mathcal{S}^T \subseteq \mathcal{T}^T$.

```
(I)  initialize                                   // register theory propagators and initialize watches
     loop
         propagate completion, loop, and recorded nogoods      // deterministically assign
         if no conflict then
             if all variables assigned then
(C)              if some δ ∈ Δ_T is violated for T ∈ 𝕋 then record δ          // check Δ_T
                 else return variable assignment                  // 𝕋-stable model found
             else
(P)              propagate theories T ∈ 𝕋     // possibly record theory nogoods from Δ_T
                 if no nogood recorded then decide        // non-deterministically assign
         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
```

Figure 1 Basic algorithm for Conflict-Driven Constraint Learning (CDCL) modulo theories

violated by $\mathbf{A}$ if $\delta \subseteq \mathbf{A}$. When $\mathbb{T} = \emptyset$, so that $\mathcal{T} = \emptyset$ as well, each total assignment $\mathbf{A}$ that does not violate any nogood $\delta \in \Delta_P \cup \Lambda_P$ yields a regular stable model of P, and such an assignment $\mathbf{A}$ is called a *solution* (for $\Delta_P \cup \Lambda_P$).

We now extend the concept of a solution to $\mathbb{T}$-stable models. To this end, we follow the idea of external propagators in [11] and identify a theory $T \in \mathbb{T}$ with a set $\Delta_T \subseteq 2^{\{\mathbf{T}a \mid a \in \mathcal{T}^T\} \cup \{\mathbf{F}a \mid a \in \mathcal{T}_e^T\}}$ of *theory nogoods* such that, given a total assignment $\mathbf{A}$, we have that $\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta_T$ iff there is no T-solution $\mathcal{S}^T$ such that $\{a \in \mathcal{T}^T \mid \mathbf{T}a \in \mathbf{A}\} \subseteq \mathcal{S}^T$ and $\{a \in \mathcal{T}_e^T \mid \mathbf{F}a \in \mathbf{A}\} \cap \mathcal{S}^T = \emptyset$. That is, the nogoods in Δ_T must reject $\mathbf{A}$ iff no T-solution (i) includes all theory atoms in $\mathcal{T}^T$ that are assigned to true by $\mathbf{A}$ and (ii) excludes all strict theory atoms in $\mathcal{T}_e^T$ assigned to false by $\mathbf{A}$. This semantic condition establishes a (one-to-one) correspondence between $\mathbb{T}$-stable models of P and solutions for $(\Delta_P \cup \Lambda_P) \cup \bigcup_{T \in \mathbb{T}} \Delta_T$. A formal elaboration can be found in [15].

The nogoods in $(\Delta_P \cup \Lambda_P) \cup \bigcup_{T \in \mathbb{T}} \Delta_T$ provide the logical fundament for the Conflict-Driven Constraint Learning (CDCL) procedure (cf. [22, 18]) outlined in Figure 1. While the completion nogoods in Δ_P are usually made explicit and subject to unit propagation, 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, i.e., when a respective propagator identifies some nogood δ such that $|\delta \setminus \mathbf{A}| \leq 1$ (and $(\{\mathbf{T}v \mid \mathbf{F}v \in \delta\} \cup \{\mathbf{F}v \mid \mathbf{T}v \in \delta\}) \cap \mathbf{A} = \emptyset$), and we say that such a nogood is *unit*. In fact, a unit nogood δ yields either a conflict, if δ is violated by $\mathbf{A}$, or otherwise a literal to be assigned by unit propagation.

While the dedicated propagator for loop nogoods is built-in in systems like *clingo* 5, those for theories are provided via the interface detailed in Section 5. To utilize custom propagators, Figure 1 includes an *initialization* step in Line (I). In addition to the "registration" of a propagator for a theory T 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 in $\mathcal{T}^T$, so that the propagator will be invoked (only) when some watched literal gets assigned.

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 in order to trigger a conflict or unit propagation, although this is not a necessary condition. The interplay of unit and theory propagation continues until a conflict or 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 $\mathbf{A}$ is total, in Line (C), theory propagators are called, one by one, for a final *check* of $\mathbf{A}$. 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 $\mathbf{A}$ 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 2 that witness the existence of a T-solution.

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 literals serves as trigger for theory propagation, theory propagators are informed when they become unassigned upon backjumping. This allows them 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.

5 Propagator Interface

We now turn to the implementation of theory propagation in *clingo* 5 and detail the structure of its interface depicted in Figure 2. 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 1. Function `Propagator.init`[4] is called once before solving (Line (I) in Figure 1) 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,[5] which are in turn associated with solver literals.[6] 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

[4] For brevity, we below drop the qualification `Propagator` and use its function names unqualified.
[5] Program literals are also used in the *aspif* format (see [15]).
[6] Note that *clasp*'s preprocessor might associate a positive or even negative solver literal with multiple atoms.

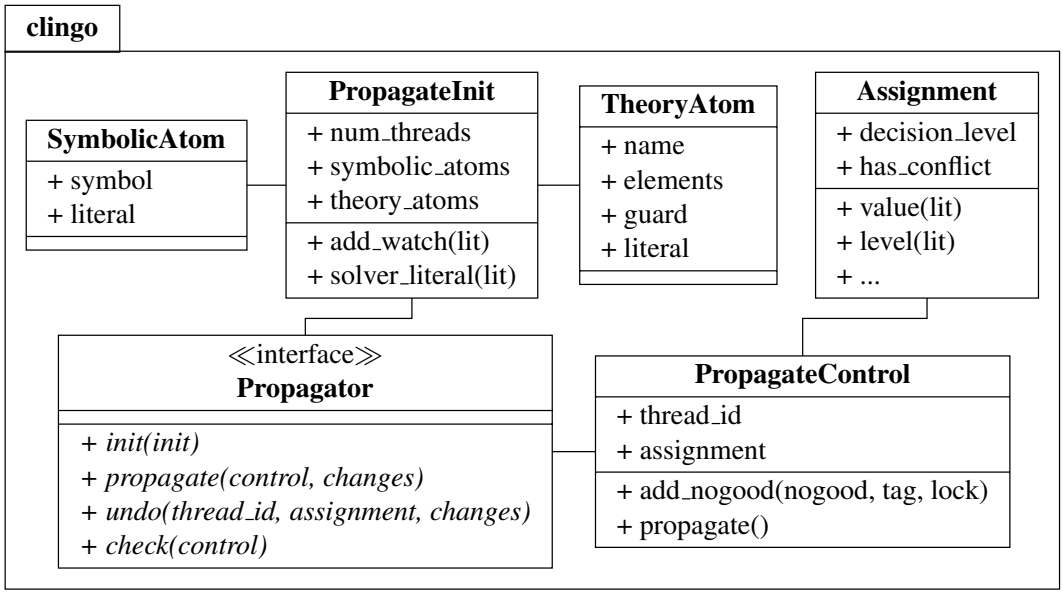

Figure 2 Class diagram of *clingo*'s (theory) propagator interface

(non-empty) list of watched literals that got assigned in the recent round of unit propagation (Line (P) in Figure 1). 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 1.[7] Function `undo` is the counterpart of `propagate` and called whenever the solver retracts assignments to watched literals (Line (U) in Figure 1). 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 1), independently of watches. Overriding the empty default implementations of propagator methods is optional. For brevity, we below focus on implementations of the methods in Python, while Lua or C could be used as well.

For illustration, consider Listing 3 giving a propagator for (half of) the pigeon-hole problem. Although this setting is constructed, it showcases central aspects that are also relevant when implementing more complex propagators, e.g., the `Pigeonator` is both stateful and can be used with multiple threads. The underlying ASP encoding is given in Line 37: A (choice) rule generates solution candidates by placing each of the p pigeons in exactly one among h holes. While the rule commented out in Line 38 would ensure that there is at most one pigeon per hole, this constraint is handled by the `Pigeonator` class implementing the `Propagator` interface (except for `check`) in Lines 8–28. Whenever two pigeons are placed in the same hole, it adds a binary nogood forbidding the placement. To this end, it maintains data structures for, given a newly placed pigeon, detecting whether there is a conflict. More precisely, the propagator has two data members: The `self.place` dictionary in Line 5 maps solver literals for `place`/2 atoms to their corresponding holes, and the `self.state` list in

[7] 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.

```
1  #script (python)
2
3  class Pigeonator:
4      def __init__(self):
5          self.place = {} # shared state
6          self.state = [] # per thread state
7
8      def init(self, init):
9          for atom in init.symbolic_atoms.by_signature("place", 2):
10             lit = init.solver_literal(atom.literal)
11             self.place[lit] = atom.symbol.args[1]
12             init.add_watch(lit)
13         self.state = [ {} for _ in range(init.num_threads) ]
14
15     def propagate(self, control, changes):
16         holes = self.state[control.thread_id]
17         for lit in changes:
18             hole = self.place[lit]
19             prev = holes.setdefault(hole, lit)
20             if prev != lit and not control.add_nogood([lit, prev]):
21                 return
22
23     def undo(self, thread_id, assignment, changes):
24         holes = self.state[thread_id]
25         for lit in changes:
26             hole = self.place[lit]
27             if holes.get(hole) == lit:
28                 del holes[hole]
29
30 def main(prg):
31     prg.register_propagator(Pigeonator())
32     prg.ground([("base", [])])
33     prg.solve()
34
35 #end.
36
37 1 { place(P,H) : H = 1..h } 1 :- P = 1..p.
38 % { place(P,H) : P = 1..p } 1 :- H = 1..h.
```

Listing 3 Propagator for the pigeon-hole problem

Line 6 stores for each solver thread its current placement of pigeons as a mapping from holes to true solver literals for `place`/2 atoms.

Function `init` in Lines 8–13 sets up watches as well as the dictionaries in `self.place` and `self.state`. To this end, it traverses (symbolic) atoms over `place`/2 in Lines 9–12. Each such atom is associated with a solver literal, obtained in Line 10. The mapping from the solver literal to its corresponding hole is then stored in the `self.place` dictionary in Line 11. In the last line of the loop, a watch is added for each solver literal at hand, so that the solver calls `propagate` whenever a pigeon is placed. Finally, in Line 13, the `self.state` list of placements per thread, subject to change upon propagation and backjumping, is initialized with empty dictionaries.

Function `propagate`, given in Lines 15–21, accesses `control.thread_id` in Line 16 to

obtain the `holes` dictionary storing the active thread's current placement of pigeons. The loop in Lines 17–21 then iterates over the list of changes, i.e., solver literals representing newly placed pigeons. After in Line 18 determining the `hole` associated with a recently assigned literal, Python's `setdefault` function is used to update the state: Depending on whether `hole` already appears as a key in the `holes` dictionary, the function either retrieves its associated literal or inserts the new literal under key `hole`. While the latter case amounts to updating the placement of pigeons, the former signals a conflict, triggered by recording a binary nogood in Line 20. Given that the solver has to resolve the conflict and backjump, the call to `add_nogood` always yields false, so that propagation stops without processing remaining changes any further.[8]

Function `undo` in Lines 23–28 resets a thread's placement of pigeons upon backjumping. Similar to `propagate`, the active thread's current placement is obtained in Line 24, and changes are traversed in Lines 25–28. The latter correspond to retracted solver literals, for which the condition in Line 27 makes sure that exactly those stored in Line 19 before are cleared, thus reflecting that the `hole` determined in Line 26 is free again. Finally, function `main` in Lines 30–33 first registers the `Pigeonator` propagator in Line 31, and then initiates grounding and solving with *clingo*.

6 Experiments

Our approach aims at a simple yet general framework for incorporating theory reasoning into ASP solving. Hence, it leaves room for various ways of encoding a problem and of implementing theory propagation. To reflect this from a practical perspective, we empirically explore several options for solving problems with difference logic (*DL*) constraints. To be more precise, we contrast an encoding relying on *defined* theory atoms with one leaving them *external* (cf. Section 3), and a *stateless* with a *stateful* propagator implementation. As a *non-strict* interpretation of *DL* constraints is sufficient for the problems given below, we stick to this option and do not vary it.

The consistency of a set C of *DL* constraints can be checked by mapping them to a weighted directed graph $G(C)$. The nodes of $G(C)$ are the (integer) variables occurring in C, and for each $x_1 - x_2 \leq k$ in C, $G(C)$ includes an edge from x_1 to x_2 with weight k. Then, C is *DL*-consistent iff $G(C)$ contains no cycle whose sum of edge weights is negative. The difference between a stateless and stateful *DL*-propagator amounts to whether the corresponding graph is built from scratch upon each invocation or only once and updated subsequently. In our experiments, we use the Bellman-Ford algorithm [6, 14] as basis for a stateless propagator, and the one in [10] for update operations in the stateful case. Both propagator implementations detect negative cycles and record (solver) literals corresponding to their weighted edges as nogoods.

Theory atoms corresponding to *DL* constraints are formed as described in Section 2. The difference between using defined and external theory atoms boils down to their occurrence in the head of a rule, as in Line 19 of Listing 1, viz.

```
&diff { end(T)-start(T) } <= D :- duration(T,D).
```

or in the body, as in

[8] The optional arguments `tag` and `lock` of `add_nogood` can be used to control the scope and lifetime of recorded nogoods. Furthermore, in a propagator that does not add violated nogoods only, function `control.propagate` can be invoked to trigger unit propagation.

Table 1 Comparison between different encodings and *DL*-propagators for scheduling problems

		ASP		ASP modulo *DL* (stateless)				ASP modulo *DL* (stateful)			
				defined		external		defined		external	
Problem	#	T	TO	T	TO	T	TO	T	TO	T	TO
Flow shop	120	569	110	283	40	382	70	**177**	**30**	281	50
Job shop	80	600	80	600	80	600	80	**37**	**0**	43	**0**
Open shop	60	405	40	214	20	213	20	**2**	**0**	**2**	**0**
Total	260	525	230	366	140	398	170	**72**	**30**	109	50

```
:- duration(T,D), not &diff { end(T)-start(T) } <= D.
```

Note that the defining usage constrains *DL*-atoms firmer than the external one: A defined *DL*-atom is true iff at least one of its bodies holds, while an external one may vary whenever its truth is *DL*-consistent yet not imposed by integrity constraints (with further problem-specific literals).

To evaluate the different options, we expressed (decision versions of) several scheduling problems [25], typically aiming at the minimization of schedules' makespan, by logic programs in the language of Section 2. *Flow shop*: A schedule corresponds to a permutation of n jobs, each including m sequential subtasks allocating machines $1, \ldots, m$ for specific amounts of time. *Job shop*: Again considering n jobs with m sequential subtasks each, where the order in which subtasks allocate machines $1, \ldots, m$ for given amounts of time is job-specific, a schedule arranges the subtasks of different jobs in one sequence per machine. *Open shop*: Given the same setting as in the job shop problem, the sequential order of the subtasks of a job is not fixed, but augments a schedule arranging the subtasks of different jobs per machine. For reasons of scalability, we refrain from optimizing the makespan of schedules, but are only interested in some feasible schedule per instance along with the corresponding earliest start times of subtasks.

The results of our experiments, run sequentially under Linux on an Intel Xeon E5520 2.27 GHz machine equipped with 24 GB main memory, are summarized in Table 1. Each *clingo* 5 run was restricted to 600 seconds wall-clock time, while memory was never exceeded. Subcolumns headed by 'T' report average runtimes, taking timeouts as 600 seconds, and those with 'TO' numbers of timeouts over '#' instances of each scheduling problem and in total. Respective results in the column headed by 'ASP' reflect the bottom-line performance obtained with plain ASP encodings, which is obviously not competitive due to the ineffectiveness of grounding problems over large numeric domains. The remaining columns consider the four combinations of encoding and *DL*-propagator features of interest. First, we observe that the stateful propagator (on the right) has a clear edge over its stateless counterpart (in the middle). Second, with both propagator implementations, the firm encoding using defined *DL*-atoms outperforms the one leaving them external on instances of the flow shop problem. While this experiment is not meant to be universal, it demonstrates that different features have an impact on the resulting performance. In how far the tuning of theory propagators matters also depends on the use case at hand, e.g., solving a challenging application problem versus rapid prototyping of dedicated reasoning procedures.

7 Discussion

The *clingo* 5 system provides a comprehensive infrastructure for enhancing ASP with theory reasoning. This ranges from generic means for expressing theories along with their support by *gringo*, over a theory-aware intermediate format, to simple yet powerful interfaces in C, Lua, and Python. In each case, a propagator can specify (up to) four basic functions to customize its integration into *clasp*'s propagation, where an arbitrary number of (independent) theory propagators can be be incorporated. Logically, ASP encodings may build upon defined or external theory atoms, and their associated conditions may be strict or non-strict. In practice, *clingo* 5 supports stateless and stateful theory propagators, which can be controlled in a fine-grained way. For instance, propagators are thread-sensitive, watches can be set to symbolic as well as theory literals, and the scope and lifetime of nogoods stemming from theory propagation can be configured.

A first step toward a more flexible ASP infrastructure was done with *clingo* 4 [16] by introducing Lua and Python APIs for multi-shot solving. Although this allows for fine-grained control of complex ASP reasoning processes, the functionality provided no access to *clasp*'s propagation and was restricted to inspecting (total) stable models. The extended framework for theory propagation relative to partial assignments (cf. Figure 1) follows the canonical approach of SMT [4]. While *dlvhex* implicitly provides access to *clasp*'s propagation, this is done on the more abstract level of higher-order logic programs. Also, *dlvhex* as well as many other systems, such as *clingcon* or *inca*, implement specialized propagation via *clasp*'s internal interfaces, whose usage is more involved and subject to change with each release. Although the new high-level interfaces may not yet fully cover all desired features, they provide a first step toward easing the development of such dedicated systems and putting them on a more stable basis. Currently, *clingo* 5's infrastructure is already used as a basis for *clingcon* 3 [3], *lc2casp* [8], and its integration with SWI-Prolog. Finally, we believe that the extended grounding capacities along with the intermediate format supplemented in [15] will also be beneficial for non-native approaches and ease the overall development of ASP-oriented solvers. This applies to systems like *dingo*, *mingo*, and *aspmt* [5], the latter implementing ASP with theory reasoning by translation to SMT, which so far had to resort to specific input formats and meta-programming to bypass the grounder.

References

1 M. Abseher, B. Bliem, G. Charwat, F. Dusberger, M. Hecher, and S. Woltran. The D-FLAT system for dynamic programming on tree decompositions. In *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*, pages 558–572. Springer, 2014.

2 M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *Proceedings of the Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*, pages 16–30, 2009.

3 M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub. Clingcon: The next generation. *Submitted for publication*, 2016.

4 C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

5 M. Bartholomew and J. Lee. System aspmt2smt: Computing ASPMT theories by SMT solvers. In *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*, pages 529–542. Springer, 2014.

6 R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

7 J. Bomanson, M. Gebser, T. Janhunen, B. Kaufmann, and T. Schaub. Answer set programming modulo acyclicity. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 143–150. Springer, 2015.

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

9 F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/ASPStandardization/`, 2012.

10 S. Cotton and O. Maler. 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)*, pages 170–183. Springer, 2006.

11 C. Drescher and T. Walsh. Answer set solving with lazy nogood generation. In *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, pages 188–200. Leibniz International Proceedings in Informatics, 2012.

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

13 T. Eiter, M. Fink, T. Krennwallner, and C. Redl. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming*, 12(4-5):659–679, 2012.

14 L. Ford and D. Fulkerson. *Flows in networks*. Princeton University Press, 1962.

15 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. Theory solving made easy with clingo 5 (extended version). Available at `http://www.cs.uni-potsdam.de/wv/publications/`, 2016.

16 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, 2014. Available at `http://arxiv.org/abs/1405.3694`.

17 M. Gebser, B. Kaufmann, R. Otero, J. Romero, T. Schaub, and P. Wanko. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, pages 350–356. AAAI Press, 2013.

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

19 T. Janhunen, G. Liu, and I. Niemelä. 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)*, pages 1–13, 2011.

20 V. Lifschitz. What is answer set programming? In *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, pages 1594–1597. AAAI Press, 2008.

21 G. Liu, T. Janhunen, and I. Niemelä. Answer set programming via mixed integer programming. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, pages 32–42. AAAI Press, 2012.

22 J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009.

23 M. Ostrowski and T. Schaub. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming*, 12(4-5):485–503, 2012.

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

25 E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.