Clingo = ASP + Control: Extended Report

Martin Gebser[1,2], Roland Kaminski[2], Benjamin Kaufmann[2], and Torsten Schaub[2]
[1]*Aalto University, Finland* [2]*University of Potsdam, Germany*

submitted [n/a]; revised [n/a]; accepted [n/a]

Abstract

We present the new ASP system *clingo* 4. Unlike its predecessors, being mere monolithic combinations of the grounder *gringo* with the solver *clasp*, the new *clingo* 4 series offers high-level constructs for realizing complex reasoning processes. Among others, such processes feature advanced forms of search, as in optimization or theory solving, or even interact with an environment, as in robotics or query-answering. Common to them is that the problem specification evolves during the reasoning process, either because data or constraints are added, deleted, or replaced. In fact, *clingo* 4 carries out such complex reasoning within a single integrated ASP grounding and solving process. This avoids redundancies in relaunching grounder and solver programs and benefits from the solver's learning capacities. *clingo* 4 accomplishes this by complementing ASP's declarative input language by control capacities expressed via the embedded scripting languages Lua and Python. On the declarative side, *clingo* 4 offers a new directive that allows for structuring logic programs into named and parameterizable subprograms. The grounding and integration of these subprograms into the solving process is completely modular and fully controllable from the procedural side, viz. the scripting languages. By strictly separating logic and control programs, *clingo* 4 also abolishes the need for dedicated systems for incremental and reactive reasoning, like *iclingo* and *oclingo*, respectively, and its flexibility goes well beyond the advanced yet still rigid solving processes of the latter.

1 Introduction

Standard Answer Set Programming (ASP; (Baral 2003)) follows a one-shot process in computing stable models of logic programs. This view is best reflected by the input/output behavior of monolithic ASP systems like *dlv* (Leone et al. 2006) and *clingo* (Gebser et al. 2011b). Internally, however, both follow a fixed two-step process. First, a grounder generates a (finite) propositional representation of the input program. Then, a solver computes the stable models of the propositional program. This rigid process stays unchanged when grounding and solving with separate systems. In fact, up to now, *clingo* provided a mere combination of the grounder *gringo* and the solver *clasp*. Although more elaborate reasoning processes are performed by the extended systems *iclingo* (Gebser et al. 2008) and *oclingo* (Gebser et al. 2011a) for incremental and reactive reasoning, respectively, they also follow a pre-defined control loop evading any user control. Beyond this, however, there is substantial need for specifying flexible reasoning processes, for instance, when it comes to interactions with an environment, as in assisted living, robotics, or with users, advanced search, as in multi-objective optimization, planning, theory solving, or heuristic search, or recurrent query answering, as in hardware analysis and testing or stream processing. Common to all these advanced forms of reasoning is that the problem specification evolves during the reasoning processes, either because data or constraints are added, deleted, or replaced.

The new *clingo* 4 series offers novel high-level constructs for realizing such complex reasoning processes. This is achieved within a single integrated ASP grounding and solving process in order to avoid redundancies in relaunching grounder and solver programs and to benefit from the

learning capacities of modern ASP solvers. To this end, *clingo* 4 complements ASP's declarative input language by control capacities expressed via the embedded scripting languages Lua and Python. On the declarative side, *clingo* 4 offers a new directive **`#program`** that allows for structuring logic programs into named and parameterizable subprograms. The grounding and integration of these subprograms into the solving process is completely modular and fully controllable from the procedural side, viz. the scripting languages embedded via the **`#script`** directive. For exercising control, the latter benefit from a dedicated *clingo* library that does not only furnish grounding and solving instructions but moreover allows for continuously assembling the solver's program in combination with the directive **`#external`**. Hence, by strictly separating logic and control programs, *clingo* 4 abolishes the need for special-purpose systems for incremental and reactive reasoning, like *iclingo* and *oclingo*, respectively, and its flexibility goes well beyond the advanced yet still rigid solving processes of the latter.

2 Background

A (normal) *rule* r is an expression of the form $a_0 \leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$, where a_i, for $0 \leq m \leq n$, is an *atom* of the form $p(t_1, \ldots, t_k)$, p/k is a predicate symbol, and $t_1, \ldots, t_k$ are terms, built from constants, variables, and functions. Letting $h(r) = a_0$, $B(r)^+ = \{a_1, \ldots, a_m\}$, and $B(r)^- = \{a_{m+1}, \ldots, a_n\}$, we also denote r by $h(r) \leftarrow B(r)^+ \cup \{{\sim}a \mid a \in B(r)^-\}$. A (normal) *logic program* P is a set of rules. We write $H(P) = \{h(r) \mid r \in P\}$ and $A(P) = H(P) \cup \bigcup_{r \in P}(B(r)^+ \cup B(r)^-)$ to denote the set of all head atoms or atoms, respectively, occurring in P. A term, atom, rule, or program is *ground* if it does not contain any variable. The Herbrand universe of P consists of all ground terms constructible from constants, at least including all integers, and function symbols in the (implicit) language of P. The *ground instance* of P, denoted by $\mathit{grd}(P)$, is the set of all ground rules constructible from rules $r \in P$ by substituting every variable in r with some element of the Herbrand universe of P. We associate P with its *positive atom dependency graph* $G(P) = (A(\mathit{grd}(P)), \{(a_0, a) \mid r \in \mathit{grd}(P), h(r) = a_0, a \in B(r)^+\})$ and call a maximal non-empty subset of $A(\mathit{grd}(P))$ inducing a strongly connected subgraph of $G(P)$ a strongly connected component of P. A set X of ground atoms is a *model* of P, if $h(r) \in X$, $B(r)^+ \not\subseteq X$, or $B(r)^- \cap X \neq \emptyset$ holds for every $r \in \mathit{grd}(P)$; X is a *stable model* of P, if X is a $\subseteq$-minimal model of $\{h(r) \leftarrow B(r)^+ \mid r \in \mathit{grd}(P), B(r)^- \cap X = \emptyset\}$.

Following (Oikarinen and Janhunen 2006), a *module* $\mathbb{P}$ is a triple (P, I, O) consisting of a ground logic program P along with sets I and O of ground *input* and *output* atoms such that $I \cap O = \emptyset$, $A(P) \subseteq I \cup O$, and $H(P) \subseteq O$. We also denote the constituents of $\mathbb{P} = (P, I, O)$ by $P(\mathbb{P}) = P$, $I(\mathbb{P}) = I$, and $O(\mathbb{P}) = O$. A set X of ground atoms is a *stable model* of a module $\mathbb{P}$, if X is a (standard) stable model of $P(\mathbb{P}) \cup \{a \leftarrow \mid a \in I(\mathbb{P}) \cap X\}$. Two modules $\mathbb{P}_1$ and $\mathbb{P}_2$ are *compositional*, if $O(\mathbb{P}_1) \cap O(\mathbb{P}_2) = \emptyset$ and, for every strongly connected component C of $P(\mathbb{P}_1) \cup P(\mathbb{P}_2)$, $O(\mathbb{P}_1) \cap C = \emptyset$ or $O(\mathbb{P}_2) \cap C = \emptyset$. Provided that $\mathbb{P}_1$ and $\mathbb{P}_2$ are compositional, their *join* is defined as the module $\mathbb{P}_1 \sqcup \mathbb{P}_2 = (P(\mathbb{P}_1) \cup P(\mathbb{P}_2), (I(\mathbb{P}_1) \setminus O(\mathbb{P}_2)) \cup (I(\mathbb{P}_2) \setminus O(\mathbb{P}_1)), O(\mathbb{P}_1) \cup O(\mathbb{P}_2))$. The module theorem (Oikarinen and Janhunen 2006) shows that a set X of ground atoms is a stable model of $\mathbb{P}_1 \sqcup \mathbb{P}_2$ iff $X = X_1 \cup X_2$ for stable models X_1 and X_2 of $\mathbb{P}_1$ and $\mathbb{P}_2$, respectively, such that $X_1 \cap (I(\mathbb{P}_2) \cup O(\mathbb{P}_2)) = X_2 \cap (I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$. For example, the modules $\mathbb{P}_1 = (\{a \leftarrow {\sim}c;\ c \leftarrow {\sim}b\}, \{b\}, \{a, c\})$ and $\mathbb{P}_2 = (\{b \leftarrow a\}, \{a\}, \{b\})$ are compositional, and combining their stable models, $\{a, b\}$ and $\{c\}$ for $\mathbb{P}_1$ as well as $\{a, b\}$ and $\emptyset$ for $\mathbb{P}_2$, yields the stable models $\{a, b\}$ and $\{c\}$ of $\mathbb{P}_1 \sqcup \mathbb{P}_2 = (P(\mathbb{P}_1) \cup P(\mathbb{P}_2), \emptyset, \{a, b, c\})$. Unlike that, $\mathbb{P}'_1 = (\{a \leftarrow b;\ c \leftarrow {\sim}a\}, \{b\}, \{a, c\})$ and $\mathbb{P}_2$ are not compositional because the

strongly connected component $\{a, b\}$ of $P(\mathbb{P}'_1) \cup P(\mathbb{P}_2)$ includes $a \in O(\mathbb{P}'_1)$ and $b \in O(\mathbb{P}_2)$. Moreover, $\{a, b\}$ is a stable model of $\mathbb{P}'_1$ and $\mathbb{P}_2$, but not of $P(\mathbb{P}'_1) \cup P(\mathbb{P}_2)$.

For associating (not necessarily ground) logic programs with modules, we adopt the respective definition from (Gebser et al. 2008). That is, for a logic program P and a set X of ground atoms, $P|_X = \{h(r) \leftarrow B(r)^+ \cup \{\sim a \mid a \in B(r)^- \cap X\} \mid r \in \mathit{grd}(P), B(r)^+ \subseteq X\}$ denotes the *projection* of P to X. Given this, the *instantiation* of P relative to a given set I of ground atoms is the module $\mathbb{P}(I) = (P|_{I \cup H(P|_X)}, I \setminus H(P|_X), H(P|_X))$, where $X = I \cup H(\mathit{grd}(P))$.

3 Controlling grounding and solving in *clingo* 4

A key feature, distinguishing *clingo* 4 from its predecessors, is the possibility to structure (non-ground) input rules into subprograms. To this end, the directive **`#program`** comes with a name and an optional list of parameters. Once given in the *clingo* 4 input, it 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, two subprograms `base` and `acid(k)` can be specified as follows:

```
1  a(1).
2  #program acid(k).
3  b(k).
4  #program base.
5  a(2).
```

Note that `base`, with an empty parameter list, is a dedicated subprogram that, in addition to rules in the scope of a directive like the one in Line 4, gathers all rules not preceded by a **`#program`** directive. Hence, in the above example, the `base` subprogram includes the facts `a(1)` and `a(2)`. Without further control instructions (see below), *clingo* 4 grounds and solves the `base` subprogram only, essentially yielding the standard behavior of ASP systems. The processing of other subprograms, such as `acid(k)` with the schematic fact `b(k)`, is subject to scripting control.

For a customized control over grounding and solving, a `main` routine (taking a control object representing the state of *clingo* 4 as argument) can be specified in either of the embedded scripting languages Lua and Python. For illustration, let us consider two Python `main` routines:

```
6  #script(python)
7  def main(prg):
8      prg.ground("base",[])
9      prg.solve()
10 #end.
```

```
6  #script(python)
7  def main(prg):
8      prg.ground("acid",[42])
9      prg.solve()
10 #end.
```

While the control program on the left matches the default behavior of *clingo* 4, the one on the right ignores all rules in the `base` program but rather, in Line 8, contains a `ground` instruction for `acid(k)`, where the parameter `k` is instantiated with the term `42`. Accordingly, the schematic fact `b(k)` is turned into `b(42)`, and the `solve` command in Line 9 yields a stable model consisting of `b(42)` only. Note that `ground` instructions apply to the subprograms given as arguments, while `solve` triggers reasoning w.r.t. all accumulated ground rules. In fact, a `solve` command makes *clingo* 4 instantiate pending subprograms and then perform reasoning. That is, when Line 9 is replaced, e.g., by `print 'Hello!'`, *clingo* 4 merely writes out `Hello!` but does neither ground any subprogram nor compute stable models.

In order to accomplish more elaborate reasoning processes, like those of *iclingo* and *oclingo* or customized ones, it is indispensable to activate or deactivate ground rules on demand. For instance, former initial or goal state conditions need to be relaxed or completely replaced when modifying a planning problem, e.g., by extending its horizon. While the predecessors of *clingo* 4

relied on a **#volatile** directive to provide a rigid mechanism for the expiration of transient rules, *clingo* 4 captures the respective functionalities and customizations thereof in terms of the directive **#external**. This directive goes back to *lparse* (Syrjänen) and was also supported by the predecessors of *clingo* 4 to exempt (input) atoms from simplifications fixing them to false. As detailed in the following, the **#external** directive of *clingo* 4 provides a generalization that, in particular, allows for a flexible handling of yet undefined atoms.

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 modules, such atoms correspond to inputs, which (unlike undefined output atoms) must not be simplified by fixing their truth value to false. In order to facilitate the declaration of input atoms, *clingo* 4 supports schematic **#external** directives that are instantiated along with the rules of their respective subprograms. To this end, a directive like

```
#external p(X,Y) : q(X,Z), r(Z,Y).
```

is treated similar to a rule r during grounding:

```
p(X,Y) :- q(X,Z), r(Z,Y).
```

However, the head atoms of resulting ground instances of r are merely collected as inputs, whereas the ground rules as such are discarded. To reflect this distinct treatment, we associate r with an annotated rule r^ϵ, in which ϵ is a distinguished atom not occurring in the *clingo* 4 input:

```
p(X,Y) :- q(X,Z), r(Z,Y), ϵ.
```

Given this, a subprogram R from the *clingo* 4 input consists of all rules r within the scope of **#program** directives with the same name and number of parameters, where `base` without parameters is used by default, along with rules r^ϵ capturing **#external** directives in the same scope.

The instantiation of a subprogram R with a list $c_1, \ldots, c_k$ of parameters, such as `acid(k)` above, relies on a list $t_1, \ldots, t_k$ of terms to replace occurrences of $c_1, \ldots, c_k$ with, both in original rules r and rules r^ϵ capturing **#external** directives in R. The parameter replacement yields a subprogram $R(c_1/t_1, \ldots, c_k/t_k)$, which is instantiated relative to inputs. For instance, providing the term `42` for parameter `k` leads to `acid(k/42)` consisting of the fact `b(42)`. However, since instances of rules r^ϵ, having ϵ in the body, are not supposed to be included in a ground program, we make use of projection to dispose of them and define $[R] = R|_{A(R)\setminus\{\epsilon\}}$.

Control instructions guiding the instantiation and assembly of subprograms can be understood in terms of modules, where `ground` instructions issued before the first or in-between two `solve` commands determine rules to instantiate and join with a module representing the previous state of *clingo* 4. The state at the beginning, i.e., before encountering the first `solve` command in a `main` routine, is simply an empty module $\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$. Then, $n \geq 0$ `ground` instructions encountered up the next, say $(i{+}1)$-th, `solve` command specify a collection $Q_{i+1} = \bigcup_{1 \leq m \leq n} R_m(c_{1_m}/t_{1_m}, \ldots, c_{k_m}/t_{k_m})$ of (non-ground) rules by providing a subprogram name R_m along with terms $t_{1_m}, \ldots, t_{k_m}$ for each $1 \leq m \leq n$. Instantiating Q_{i+1} with input and output atoms from the previous state, viz. $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$, along with the distinguished atom ϵ yields a module $\mathbb{Q}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i) \cup \{\epsilon\})$. As its ground program may include instances of rules r^ϵ stemming from **#external** directives, we must drop the input atom ϵ and instances of rules r^ϵ. At the same time, we need to reinterpret head atoms defined by dropped instances of rules r^ϵ as inputs. To this end, we define the module $[\mathbb{Q}] = ([Q], (I \setminus \{\epsilon\}) \cup (H(Q) \setminus H([Q])), O \setminus (H(Q) \setminus H([Q])))$, where $Q = P(\mathbb{Q})$, $I = I(\mathbb{Q})$, and $O = O(\mathbb{Q})$ for $\mathbb{Q} = \mathbb{Q}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i) \cup \{\epsilon\})$. In fact, $H(Q) \setminus H([Q])$ contains the atoms stemming from **#external** directives. Provided that the module $\mathbb{P}_i$, representing the previous state of *clingo* 4, and $[\mathbb{Q}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i) \cup \{\epsilon\})]$ are

compositional, their join $\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup [\mathbb{Q}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i) \cup \{\epsilon\})]$ captures the state of *clingo* 4 at the $(i+1)$-th `solve` command in a `main` routine. When solving with $\mathbb{P}_{i+1}$, the input atoms in $I(\mathbb{P}_{i+1})$ are taken to be false by default, which can still be altered as illustrated next.

Let us demonstrate the formation of modules along with input atoms on the following example:

```
1  #external p(1;2;3).
2  p(0) :- p(3).
3  p(0) :- not p(0).

4  #program succ(n).
5  #external p(n+3).
6  p(n) :- p(n+3).
7  p(n) :- not p(n+1), not p(n+2).

8  #script(python)
9  from gringo import Fun
10 def main(prg):
11     prg.ground("base", [])
12     prg.assignExternal(Fun("p", [3]), True)
13     prg.solve()
14     prg.assignExternal(Fun("p", [3]), False)
15     prg.solve()
16     prg.ground("succ", [1])
17     prg.ground("succ", [2])
18     prg.solve()
19     prg.ground("succ", [3])
20     prg.solve()
21 #end.
```

In view of the `ground` instruction in Line 11, issued before the first `solve` command in Line 13, rules and **`#external`** directives from the `base` subprogram in Line 1–3 yield the instantiation:

$$\mathbb{Q}_1(\{\epsilon\}) = \left(\left\{\begin{matrix} p(0) \leftarrow p(3);\ p(0) \leftarrow \sim p(0); \\ p(1) \leftarrow \epsilon;\ p(2) \leftarrow \epsilon;\ p(3) \leftarrow \epsilon \end{matrix}\right\}, \{\epsilon\}, \{p(0), p(1), p(2), p(3)\}\right)$$

Joining $[\mathbb{Q}_1(\{\epsilon\})]$ with $\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$ then leads to the following first state of *clingo* 4:

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup [\mathbb{Q}_1(\{\epsilon\})] = (\{p(0) \leftarrow p(3);\ p(0) \leftarrow \sim p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

While the input atoms $p(1)$ and $p(2)$ are (by default) assigned to false when solving $\mathbb{P}_1$, the instruction in Line 12 switches the value of $p(3)$ to true, so that the stable model $\{p(0), p(3)\}$ of $\mathbb{P}_1$ is obtained in Line 13. Next, the instruction in Line 14 turns $p(3)$ to false, and since no further subprograms are instantiated, no stable model is obtained when solving $\mathbb{P}_2 = \mathbb{P}_1 \sqcup (\emptyset, \{p(0), p(1), p(2), p(3)\}, \emptyset) = \mathbb{P}_1$ in view of the `solve` command in Line 15. Afterwards, the `ground` instructions in Line 16 and 17 express that rules and **`#external`** directives from `succ(n/1)` and `succ(n/2)`, replacing the parameter of subprogram `succ(n)` in Line 5–7 with terms, are to be instantiated relative to the atoms $I_3 = \{p(0), p(1), p(2), p(3), \epsilon\}$, which leads to:

$$\mathbb{Q}_3(I_3) = \left(\left\{\begin{matrix} p(1) \leftarrow p(4);\ p(1) \leftarrow \sim p(2), \sim p(3);\ p(4) \leftarrow \epsilon; \\ p(2) \leftarrow p(5);\ p(2) \leftarrow \sim p(3), \sim p(4);\ p(5) \leftarrow \epsilon \end{matrix}\right\}, \left\{\begin{matrix} p(0), \\ p(3), \epsilon \end{matrix}\right\}, \left\{\begin{matrix} p(1), p(2), \\ p(4), p(5) \end{matrix}\right\}\right)$$

This yields the join $\mathbb{P}_3 = \mathbb{P}_2 \sqcup [\mathbb{Q}_3(I)]$ of $[\mathbb{Q}_3(I)]$ and $\mathbb{P}_2 = \mathbb{P}_1$ from the previous *clingo* 4 state:

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(1) \leftarrow p(4);\ p(1) \leftarrow \sim p(2), \sim p(3);\ p(0) \leftarrow p(3); \\ p(2) \leftarrow p(5);\ p(2) \leftarrow \sim p(3), \sim p(4);\ p(0) \leftarrow \sim p(0) \end{matrix}\right\}, \left\{\begin{matrix} p(3), \\ p(4), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(0), \\ p(1), p(2) \end{matrix}\right\}\right)$$

Since none of the input atoms in $I(\mathbb{P}_3)$ is set to true at the `solve` command in Line 18, solving with $\mathbb{P}_3$ gives no stable model. Then, *clingo* 4 proceeds by instantiating `succ(n/3)` relative to $I_4 = \{p(0), p(1), p(2), p(3), p(4), p(5), \epsilon\}$ in view of the `ground` instruction in Line 19:

$$\mathbb{Q}_4(I_4) = \left(\left\{ \begin{array}{l} p(3) \leftarrow p(6);\ p(6) \leftarrow \epsilon; \\ p(3) \leftarrow \sim p(4), \sim p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(1), p(2), \\ p(4), p(5), \epsilon \end{array} \right\}, \left\{ \begin{array}{l} p(3), \\ p(6) \end{array} \right\} \right)$$

Finally, the `solve` command in Line 20 leads to the stable model $\{p(0), p(3)\}$ of the following module $\mathbb{P}_4 = \mathbb{P}_3 \sqcup [\mathbb{Q}_4(I_4)]$ capturing the residual *clingo* 4 state:

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{l} p(1) \leftarrow p(4);\ p(1) \leftarrow \sim p(2), \sim p(3);\ p(0) \leftarrow p(3); \\ p(2) \leftarrow p(5);\ p(2) \leftarrow \sim p(3), \sim p(4);\ p(0) \leftarrow \sim p(0); \\ p(3) \leftarrow p(6);\ p(3) \leftarrow \sim p(4), \sim p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(4), \\ p(5), \\ p(6) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(1), \\ p(2), p(3) \end{array} \right\} \right)$$

The above example illustrates the customized selection of (non-ground) subprograms to instantiate in-between `solve` commands. For a convenient declaration of input atoms from other subprogram instances, schematic **`#external`** directives are embedded into the grounding process. Given that they do not contribute ground rules, but merely qualify (undefined) atoms that should be exempted from simplifications, **`#external`** directives address the signature of subprograms' ground instances. Hence, it is advisable to condition them by domain predicates[1] (Syrjänen) only, as this precludes any interferences between signatures and grounder implementations. As long as input atoms remain undefined, their truth values can be freely picked and modified in-between `solve` commands via `assignExternal` instructions, which thus allow for configuring the inputs to modules representing *clingo* 4 states in order to select among their stable models. Unlike that, the predecessors *iclingo* and *oclingo* of *clingo* 4 always assigned input atoms to false, so that the addition of rules was necessary to accomplish switching truth values as in Line 12 and 14 above. However, for a well-defined semantics, *clingo* 4 like its predecessors builds on the assumption that the modules induced by subprograms' instantiations are compositional, which essentially requires definitions of (head) atoms and mutual positive dependencies to be local to evolving ground programs (cf. (Gebser et al. 2008)).

4 Using *clingo* 4 in practice

After describing the general grounding and solving process of *clingo* 4, we now illustrate its advanced features and application scenarios. To begin with, we consider the well-known n-queens problem, aiming at a reasoning process solving series of boards with different size. Given that larger boards subsume smaller ones, an evolving problem specification can reuse ground rules from previous *clingo* 4 states when the size increases. To this end, we view the increment of n by one as the addition of one more row and column. The basic idea is to interconnect the previous and added board cells such that any of them has a unique predecessor or successor in either of the four attack directions of queens. The respective connection schemes are depicted in Figure 1(a)–(d), where direct links are indicated by arrows to target cells with a (white or black) circle. The scheme for backward diagonals, displayed in Figure 1(a), connects cells of the uppermost previous row to corresponding attacked cells in a new column; the latter are in turn linked to the new cells they attack in the row above. Note that this scheme ensures that, starting from the middle of any backward diagonal, all cells that are successively added are on a path. Such a path follows

[1] Domain and built-in predicates have unique extensions that can be evaluated entirely by means of grounding.

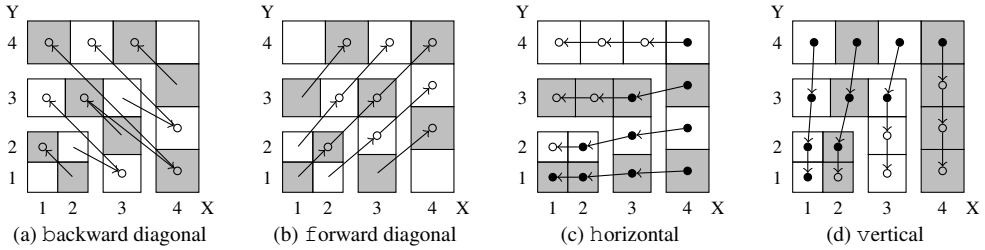

Fig. 1: Attack target links among cells of successive n-queens boards up to size 4

the board evolution and is directed from previous to newly added cells, where white circles in arrow targets indicate the presence of attacking cells on the board when their respective target cells are added. The schemes for attacks along forward diagonals, horizontal rows, and vertical columns are shown in Figure 1(b), 1(c), and 1(d). Notably, the latter two (partially) link new cells to previous ones, in which case the targets are highlighted by black circles. At the level of modules, links from cells that may be added later on give rise to input atoms.

The *clingo* 4 encoding in Listing 1 is based on the links depicted in Figure 1(a)–(d). After declaring `queen/2` as the output predicate to be displayed, the (sub)program `board(n)` provides rules for extending a board of size $n-1$ to $n \geq 1$. To this end, the **`#external`** directives in Line 4 and 5 declare atoms representing horizontal and vertical attacks on cells in the n-th column or row, respectively, as inputs. Such atoms match the targets of arrows leading to cells with black circles in Figure 1(c) and 1(d). For instance, `attack(2,1,h)` and `attack(2,2,h)` as well as `attack(1,2,v)` and `attack(2,2,v)` are the inputs to `board(n/2)`, expressing that cells at the horizontal and vertical borders can become targets of attacks once the board is extended beyond size 2. The instances of `target(X,Y,X',Y',D,n)` specified in Line 7–13 provide links from cells `(X,Y)` to targets `(X',Y')` along with directions `D` leading from or to some newly added cell in the n-th column or row. These instances correspond to arrows shown in Figure 1(a)–(d), yet omitting those to border cells such that `attack(X',Y',D)` is declared as input in Line 4 and 5, also highlighted by black circles in Figure 1(c) and 1(d). Queens at newly added cells in the n-th column or row are enabled via the choice rule in Line 15, and the links provided by instances of `target(X,Y,X',Y',D,n)` are utilized in Line 17 and 18 for deriving `attack(X',Y',D)` in view of a queen at cell `(X,Y)` or any of its predecessors in the direction indicated by `D`. For instance, the following ground rules, simplified by dropping atoms of the domain predicate `target/6`, capture horizontal attacks along the first row of a board of size 4:

```
attack(1,1,h) :- queen(2,1).    attack(1,1,h) :- attack(2,1,h).
attack(2,1,h) :- queen(3,1).    attack(2,1,h) :- attack(3,1,h).
attack(3,1,h) :- queen(4,1).    attack(3,1,h) :- attack(4,1,h).
```

Note that a queen represented by an instance of `queen(X,1)`, for $2 \leq$ `X` ≤ 4, propagates to cells on its left via an implication chain deriving `attack(X',1,h)` for every $1 \leq$ `X'` $<$ `X`. Moreover, the fact that the cell at `(4,1)` can be attacked from the right when increasing the board size is reflected by the input atom `attack(4,1,h)` declared in `board(n/4)`. Given that attacks are propagated analogously for other rows and directions, instances of the integrity constraint in Line 20 prohibit a queen at cell `(X',Y')` whenever `attack(X',Y',D)` signals that some predecessor in either direction `D` has a queen already. The integrity constraints in Line 22 and 23 additionally require that each row and column contains some queen. In view of the orientations

```
1  #show queen/2.

3  #program board(n).
4  #external attack(n,1..n,h).
5  #external attack(1..n,n,v).

7  target(n,  X,  X,  n,  b,n) :- X = 1..n-1.              % diagonal b
8  target(Y,  n-1,n,  Y-1,b,n) :- Y = 2..n-1.              % diagonal b
9  target(X,  n-1,X+1,n,  f,n) :- X = 1..n-1.              % diagonal f
10 target(n-1,Y,  n,  Y+1,f,n) :- Y = 1..n-2.              % diagonal f
11 target(X,  n,  X-1,n,  h,n) :- X = 2..n.                % horizontal
12 target(n,  Y,  n-1,Y,  h,n) :- Y = 1..n-1.              % horizontal
13 target(Y,  X,  Y,  X-1,v,n) :- target(X,Y,X-1,Y,h,n).   % vertical

15 { queen(1..n,n); queen(n,1..n-1) }.

17 attack(X',Y',D) :- target(X,Y,X',Y',D,n), queen(X,Y).
18 attack(X',Y',D) :- target(X,Y,X',Y',D,n), attack(X,Y,D).

20 :- target(X,Y,X',Y',D,n), attack(X',Y',D), queen(X',Y').

22 :- not queen(1,n), not attack(1,n,h).
23 :- not queen(n,1), not attack(n,1,v).

25 #script(python)
26 def main(prg):
27     n = 0
28     for lower, upper in prg.getConst("calls").args():
29         while n < upper:
30             n += 1
31             prg.ground("board", [n])
32             if n >= lower:
33                 print 'SIZE {0}'.format(n)
34                 prg.solve()
35 #end.
```

Listing 1: *clingo* 4 program for successive n-queens solving (`queens.lp`)

of horizontal and vertical links, as displayed in Figure 1(c) and 1(d), non-emptiness can be recognized from a queen at or an attack propagated to the first position in a row or column, no matter to which size the board is extended later on. Importantly, instantiations of `board(n)` with different terms, i.e., integers, for `n` define distinct (ground) atoms, and the non-circularity of paths according to the connection schemes in Figure 1(a)–(d) excludes mutual positive dependencies (between instances of `attack(X',Y',D)`). Hence, the modules induced by different instantiations of `board(n)` are compositional and can be joined to successively increase the board size.

The Python `main` routine in Line 25–35 of Listing 1 implements control for the successive grounding and solving of a series of boards. To this end, an ordered list of integer intervals is to be provided on the command-line, e.g., `-c calls="list((1,1),(3,5),(8,9))"`. As long as the upper limit of some interval is yet unreached, the board size is incremented by one in Line 30 and, in view of the `ground` instruction in Line 31, taken as a term for instantiating `board(n)`. However, solving is only invoked in Line 34 if the current size lies within the interval

```
1  #script(python)
2  from gringo import Fun, SolveResult

4  def init(val, default):
5      return val if val != None else default

7  def main(prg):
8      stop = str(init(prg.getConst("istop"), "SAT"))
9      step = int(init(prg.getConst("iinit"), 0))

11     prg.ground("base", [])
12     while True:
13         step += 1
14         prg.ground("cumulative", [step])
15         prg.assignExternal(Fun("query", [step]), True)
16         print 'STEP {0}'.format(step)
17         ret = prg.solve()
18         if (stop == "SAT"   and ret == SolveResult.SAT) or \
19            (stop == "UNSAT" and ret == SolveResult.UNSAT): break
20         prg.releaseExternal(Fun("query", [step]))
21 #end.
```

Listing 2: Python script implementing *iclingo* functionality in *clingo* (`iclingo.lp`)

of interest. Provided that this is the case for any particular $n \geq 1$, the sequence of issued `ground` instructions makes sure that the current *clingo* 4 state corresponds to the module obtained by instantiating and joining the subprograms `board(n/`i`)`, for $1 \leq i \leq n$, in increasing order. Since all ground rules accumulated in such a state are relevant (and not superseded by permanently falsifying the body) for n-queens solving, there is no redundancy in instantiating `board(n/`i`)` for each $1 \leq i \leq n$, even when the provided integer intervals do not include i and i-queens solving is skipped. For instance, `-c calls="list((1,1),(3,5),(8,9))"` specifies a series of six boards to solve, while the subprogram `board(n)` is successively instantiated with nine different terms for parameter `n`. In fact, the `main` routine in Line 25–35 automates the assembly of subprograms needed to process an arbitrary yet increasing sequence of board sizes.

As mentioned above, *clingo* 4 fully supersedes its special-purpose predecessors *iclingo* and *oclingo*. To illustrate this, we give in Listing 2 a slightly simplified version of *iclingo*'s control loop in Python. The full control loop (included in the release) mainly adds handling of further *iclingo* options. Roughly speaking, *iclingo* offers a step-oriented, incremental approach to ASP that avoids redundancies by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem (as in iterative deepening search). 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`. These parts were delineated in *iclingo* by the directives **`#base`**, **`#cumulative t`**, and **`#volatile t`**. In *clingo* 4, all three directives are captured by **`#program`** declarations along with **`#external`** for volatile rules.

We illustrate this by adapting the Towers of Hanoi encoding from (Gebser et al. 2012) in Figure 2. The problem instance in Figure 2(a) as well as Line 2 in 2(b) constitute static knowledge and thus belong to the base part. The transition function is described in the cumulative part in Line 5–15 of Figure 2(b). Finally, the query is expressed in Line 18; its volatility is realized

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

(a) Towers of Hanoi instance

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

4  #program cumulative(t).
5  1 { move(D,P,t) : disk(D), peg(P) } 1.

7  move(D,t)        :- move(D,P,t).
8  on(D,P,t)        :- move(D,P,t).
9  on(D,P,t)        :- on(D,P,t-1), not move(D,t).
10 blocked(D-1,P,t) :- on(D,P,t-1).
11 blocked(D-1,P,t) :- blocked(D,P,t), disk(D).

13 :- move(D,P,t), blocked(D-1,P,t).
14 :- move(D,t), on(D,P,t-1), blocked(D,P,t).
15 :- disk(D), not 1 { on(D,P,t) } 1.

17 #external query(t).
18 :- query(t), goal_on(D,P), not on(D,P,t).
```

(b) Towers of Hanoi incremental encoding

Fig. 2: Towers of Hanoi instance (`tohI.lp`) and incremental encoding (`tohE.lp`)

by making the actual goal condition `goal_on(D,P), not on(D,P,t)` subject to the truth assignment to the external atom `query(t)`. Grounding and solving of the program in Figure 2(a) and 2(b) is controlled by the Python script in Listing 2. Line 4–9 fix the `stop` criterion and initial value of the `step` variable. Both can be supplied as constants `istop` and `iinit` when invoking *clingo* 4. Once the base part is grounded in Line 11, the script loops until the `stop` criterion is met in Line 18–19. In each iteration, the current value of `step` is used in Line 14 and 15 to instantiate the subprogram `cumulative(t)` and to set the respective external atom `query(t)` to true. If the `stop` condition is yet unfulfilled w.r.t. the result of solving the extended program, the current `query(t)` atom is permanently falsified (cf. Line 17–20), thus annulling the corresponding instances of the integrity constraint in Line 18 of Figure 2(b) before they are replaced in the next iteration.

Another innovative feature of *clingo* 4 is its incremental optimization. This allows for adapting objective functions along the evolution of a program at hand. A simple example is the search for shortest plans when increasing the horizon in non-consecutive steps. To see this, recall that literals in minimize statements (and analogously weak constraints) are supplied with a sequence of terms of the form $w@p, \vec{t}$, where w and p are integers providing a weight and a priority level and $\vec{t}$ is a sequence of terms (cf. (Calimeri et al. 2012)). As an example, consider the subprogram:

```
#program cumulativeObjective(t).
#minimize{ W@P,X,Y,t : move(X,Y,W,P,t) }.
% or :~ move(X,Y,W,P,t). [W@P,X,Y,t]
```

When grounding and solving `cumulativeObjective(t)` for successive values of `t`, the solver's objective function (per priority level `P`) is gradually extended with new atoms over `move/5`, and all previous ones are kept.

Moreover, for enabling the removal of literals from objective functions, we can use externals:

```
#program volatileObjective(t).
```

```
#external activateObjective(t).
#minimize{ W@P,X,Y,t : move(X,Y,W,P,t), activateObjective(t) }.
```

The subprogram `volatileObjective(t)` behaves like `cumulativeObjective(t)` as long as the external atom `activateObjective(t)` is true. Once it is set to false, all atoms over `move/5` with the corresponding term for `t` are dismissed from objective functions.

A reasoning process in *clingo* 4 is partitioned into a sequence of solver invocations. We have seen how easily the solver's logic program can be altered at each step. Sometimes it is useful to do this in view of a previously obtained stable model. For this purpose, the `solve` command can be equipped with an (optional) callback function `onModel`. For each stable model found during a call to `solve(onModel)`, an object encompassing the model is passed to `onModel`, whose implementation can then access and inspect the model. A typical example is the addition of constraints based on the last model that are then supplied to the solver before computing the next one. An application is theory solving by passing (parts of) the last model to a theory solver for theory-based consistency checking or for providing the value of an externally evaluated objective function. Moreover, *clingo* 4 also furnishes an asynchronous solving function `asolve` that launches an interruptable solving process in the background. This is particularly useful in reactive settings in order to stop solving upon the arrival of new external information.

Similarly, the configuration of *clasp* can be changed at each step via the function `setConf`, taking a string including command line options along with a flag indicating whether the previous configuration is updated or replaced as arguments. For instance, this allows for changing search parameters, reasoning modes, number of threads, etc. Changing search parameters is of interest when addressing computational tasks involving the generation of several models, like optimal planning, multi-criteria optimization, or heuristic search. Apart from analyzing the previous model via the `onModel` callback, one can also monitor the search progress by means of the function `getStats`, returning an object encapsulating up to 135 attributes of the previous search process. Furthermore, *clingo* 4 allows for customizing the heuristic values of variables, as described in (Gebser et al. 2013a). At a higher level, a user may simply want to explore the set of models, and decide to compute first one, then all, and then the intersection or union of all models. This can be interleaved with the addition of subprograms via the function `add`, which may in turn include **`#external`** directives to declare temporary hypotheses. The experienced reader may note that this can be done fully interactively by means of IPython. Practical examples for the mentioned features can be found in the releases at (potassco).

5 Related work

Although *clingo* 3 (Gebser et al. 2011c) already featured Lua as an embedded scripting language, its usage was limited to (deterministic) computations during grounding; neither were library functions furnished by *clingo* 3.

Of particular interest is *dlvhex* (Fink et al. 2013), an ASP system aiming at the integration of external computation sources. For this purpose, *dlvhex* relies on higher-order logic programs using external higher-order atoms for software interoperability. Such external atoms should not be confused with *clingo*'s **`#external`** directive because they are evaluated via procedural means during solving. Given this, *dlvhex* can be seen as an *ASP modulo Theory* solver, similar to SAT modulo Theory solvers (Nieuwenhuis et al. 2006). In fact, *dlvhex* uses *gringo* and *clasp* as backends and follows the design of the *ASP modulo CSP* solver *clingcon* (Ostrowski and Schaub 2012) in communicating with external "oracles" through *clasp*'s post propagation mechanism.

In this way, theory solvers are tightly integrated into the ASP system and have access to the solver's partial assignments. Unlike this, the light-weighted theory solving approach offered by *clingo* 4 can only provide access to total (stable) assignments. It is thus interesting future work to investigate in how far *dlvhex* can benefit from lifting its current low-level integration into *clasp* to a higher level in combination with *clingo* 4. Clearly, the above considerations also apply to extensions of *dlvhex*, such as *acthex* (Fink et al. 2013). Furthermore, *jdlv* (Febbraro et al. 2012) encapsulates the *dlv* system to facilitate one-shot ASP solving in Java environments by providing means to generate and process logic programs, and to afterwards extract their stable models.

The procedural attachment to the *idp* system (De Pooter et al. 2013; De Cat et al. 2014) builds on interfaces to C++ and Lua. Like *clingo* 4, it allows for evaluating functions during grounding, calling the grounder and solver multiple times, inspecting solutions, and reacting to external input after search. The emphasis, however, lies on high-level control blending in with *idp*'s modeling language, while *clingo* 4 offers more fine-grained control over the grounding and solving process, particularly aiming at a flexible incremental assembly of programs from subprograms.

In SAT, incremental solver interfaces from low-level APIs are common practice. Pioneering work was done in *minisat* (Eén and Sörensson 2004), furnishing a C++ interface for solving under assumptions. In fact, the *clasp* library underlying *clingo* 4 builds upon this functionality to implement incremental search (see (Gebser et al. 2008)). Given that SAT deals with propositional formulas only, solvers and their APIs lack support for modeling languages and grounding. Unlike this, the SAT modulo Theory solver *z3* (de Moura and Bjørner 2008) comes with a Python API that, similar to *clingo* 4, provides a library for controlling the solver as well as language bindings for constraint handling. In this way, Python can be used as a modeling language for *z3*.

6 Discussion

The new *clingo* 4 system complements ASP's declarative input language by control capacities expressed by embedded scripting languages. This is accomplished within a single integrated ASP grounding and solving process in which a logic program may evolve over time. The addition, deletion, and replacement of programs is controlled procedurally by means of *clingo*'s dedicated library. The incentives for evolving a logic program are manifold and cannot be captured with the standard one-shot approach of ASP. Examples include unrolling a transition function, as in planning, interacting with an environment, as in assisted living, robotics, or stream reasoning, interacting with a user exploring a domain, theory solving, and advanced forms of search. Addressing these demands by embedded scripting languages provides us with a generic and transparent approach. Unlike this, previous systems, like *iclingo* and *oclingo*, had a dedicated purpose involving rigid control capacities buried in monolithic programs. Rather than that, the basic technology of *clingo* 4 allows us to instantiate subprograms in-between solver invocations in a fully customizable way. On the declarative side, the availability of program parameters and the embedding of `#external` directives into the grounding process provide great flexibility in modeling schematic subprograms. In addition, the possibility of assigning input atoms facilitates the implementation of applications such as query answering or sliding window reasoning, as truth values can now be switched without manipulating a logic program.

The semantic underpinnings of our framework in terms of module theory capture the dynamic combination of logic programs in a generic way. It is interesting future work to investigate how dedicated change operations whose interest was so far mainly theoretic, like updating (Alferes et al. 2002) or forgetting (Zhang and Foo 2006), can be put into practice within this framework.

The input language of *clingo* 4 extends the *ASP-Core-2* standard (Calimeri et al. 2012). Although we have presented *clingo* 4 for normal logic programs, we mention that it accepts (extended) disjunctive logic programs, processed via the multi-threaded solving approach described in (Gebser et al. 2013b). In version 4.3, *clingo* moreover embeds *clasp* 3, featuring domain-specific heuristics (Gebser et al. 2013a) and optimization using unsatisfiable cores (Andres et al. 2012). *clingo* 4 is freely available at (potassco), and its releases include many best practice examples illustrating the aforementioned application scenarios.

Acknowledgments This work was partially funded by DFG grant SCHA 550/9-1.

References

ALFERES, J., PEREIRA, L., PRZYMUSINSKA, H., AND PRZYMUSINSKI, T. 2002. LUPS: A language for updating logic programs. *Artificial Intelligence 138,* 1-2, 87–116.

ANDRES, B., KAUFMANN, B., MATHEIS, O., AND SCHAUB, T. 2012. Unsatisfiability-based optimization in clasp. In *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, A. Dovier and V. Santos Costa, Eds. Leibniz International Proceedings in Informatics, vol. 17. Dagstuhl Publishing, 212–221.

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

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., RICCA, F., AND SCHAUB, T. 2012. ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.0.pdf`.

DE CAT, B., BOGAERTS, B., BRUYNOOGHE, M., AND DENECKER, M. 2014. Predicate logic as a modelling language: The IDP system. *CoRR abs/1401.6312*. Available at `http://arxiv.org/abs/1401.6312v1`.

DE MOURA, L. AND BJØRNER, N. 2008. Z3: An efficient SMT solver. In *Proceedings of the Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, C. Ramakrishnan and J. Rehof, Eds. Lecture Notes in Computer Science, vol. 4963. Springer-Verlag, 337–340.

DE POOTER, S., WITTOCX, J., AND DENECKER, M. 2013. A prototype of a knowledge-based programming environment. In *Proceedings of the Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management (INAP'11) and the Twenty-fifth Workshop on Logic Programming (WLP'11)*, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 7773. Springer-Verlag, 279–286.

DELGRANDE, J. AND FABER, W., Eds. 2011. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. Lecture Notes in Artificial Intelligence, vol. 6645. Springer-Verlag.

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. Lecture Notes in Computer Science, vol. 2919. Springer-Verlag, 502–518.

FEBBRARO, O., LEONE, N., GRASSO, G., AND RICCA, F. 2012. JASP: A framework for integrating answer set programming with Java. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, G. Brewka, T. Eiter, and S. McIlraith, Eds. AAAI Press, 541–551.

FINK, M., GERMANO, S., IANNI, G., REDL, C., AND SCHÜLLER, P. 2013. ActHEX: Implementing HEX programs with action atoms. In *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, P. Cabalar and T. Son, Eds. Lecture Notes in Artificial Intelligence, vol. 8148. Springer-Verlag, 317–322.

GEBSER, M., GROTE, T., KAMINSKI, R., AND SCHAUB, T. 2011a. Reactive answer set programming. See Delgrande and Faber (2011), 54–66.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND SCHNEIDER, M. 2011b. 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. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, 190–205.

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., KÖNIG, A., AND SCHAUB, T. 2011c. Advances in gringo series 3. See Delgrande and Faber (2011), 345–351.

GEBSER, M., KAUFMANN, B., OTERO, R., ROMERO, J., SCHAUB, T., AND WANKO, P. 2013a. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, M. desJardins and M. Littman, Eds. AAAI Press, 350–356.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2013b. Advanced conflict-driven disjunctive answer set solving. In *Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13)*, F. Rossi, Ed. IJCAI/AAAI Press, 912–918.

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.

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.

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.

OSTROWSKI, M. AND SCHAUB, T. 2012. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming 12,* 4-5, 485–503.

POTASSCO. `http://potassco.sourceforge.net`.

SYRJÄNEN, T. Lparse 1.0 user's manual. Available at `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

ZHANG, Y. AND FOO, N. 2006. Solving logic program conflict through strong and weak forgettings. *Artificial Intelligence 170,* 8-9, 739–778.

Appendix A Running *clingo* 4 on n-Queens example

```
$ clingo queens.lp -c calls="list((1,1),(3,5),(8,9))"
clingo version 4.3.0
Reading from queens.lp
SIZE 1
Solving...
Answer: 1
queen(1,1)
SIZE 3
Solving...
SIZE 4
Solving...
Answer: 1
queen(2,1) queen(1,3) queen(4,2) queen(3,4)
SIZE 5
Solving...
Answer: 1
queen(2,1) queen(3,3) queen(1,4) queen(5,2) queen(4,5)
SIZE 8
Solving...
Answer: 1
queen(2,1) queen(1,4) queen(5,2) queen(3,5) queen(7,3) \
queen(6,7) queen(8,6) queen(4,8)
SIZE 9
Solving...
Answer: 1
queen(3,2) queen(1,3) queen(6,4) queen(5,6) queen(7,1) \
queen(2,7) queen(8,5) queen(4,8) queen(9,9)
SATISFIABLE

Models       : 5+
Calls        : 6
Time         : 0.031s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.020s
```

Appendix B Running *clingo* 4 on Towers of Hanoi example

```
$ clingo iclingo.lp tohI.lp tohE.lp
clingo version 4.3.0
Reading from iclingo.lp ...
STEP 1
Solving...
STEP 2
Solving...
STEP 3
Solving...
STEP 4
Solving...
STEP 5
Solving...
STEP 6
Solving...
STEP 7
Solving...
STEP 8
Solving...
STEP 9
Solving...
STEP 10
Solving...
STEP 11
Solving...
STEP 12
Solving...
STEP 13
Solving...
STEP 14
Solving...
STEP 15
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        : 15
Time         : 0.042s (Solving: 0.01s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.030s
```