Ricochet Robots Reloaded
A Case-study in Multi-shot ASP Solving

Martin Gebser[1 2], Roland Kaminski[1], Philipp Obermeier[1], and Torsten Schaub[1 3 ⋆]

[1] Aalto University, Finland
[2] University of Potsdam, Germany
[3] Inria Rennes, France

Abstract. Nonmonotonic reasoning is about drawing conclusions in the absence of (complete) information. Hence, whenever new information arrives, one may have to withdraw previously drawn conclusions. In fact, Answer Set Programming is nowadays regarded as the computational embodiment of nonmonotonic reasoning. However, traditional answer set solvers do not account for changing information. Rather they are designed as one-shot solvers that take a logic program and compute its stable models, basta! When new information arrives the program is extended and the solving process is started from scratch once more. Hence the dynamics giving rise to nonmonotonicity is not reflected by such solvers and left to the user. This shortcoming is addressed by multi-shot solvers that embrace the dynamicity of nonmonotonic reasoning by allowing a reactive procedure to loop on solving while acquiring changes in the problem specification.
In this paper, we provide a hands-on introduction to multi-shot solving with *clingo* 4 by modeling the popular board game of *Ricochet Robots*. Our particular focus lies on capturing the underlying turn based playing through the procedural-declarative interplay offered by the Python-ASP integration of *clingo* 4. From a technical perspective, we provide semantic underpinnings for multi-shot solving with *clingo* 4 by means of a simple stateful semantics along with operations reflecting *clingo* 4 functionalities.

1 Introduction

Nonmonotonic reasoning [1, 2] is about drawing conclusions in the absence of (complete) information. Hence, whenever new information arrives, one may have to withdraw previously drawn conclusions. In fact, Answer Set Programming (ASP; [3, 4]) can nowadays be regarded as the computational embodiment of nonmonotonic reasoning. However, traditional ASP solvers do not account for changing information. Rather they are designed as one-shot solvers that take a logic program and compute its stable models, basta! When new information arrives the program is extended and the solving process is re-started from scratch again. Hence, the dynamics giving rise to nonmonotonicity is not reflected by such solvers and left to the user. Turning towards the future, ASP is and will be an under-the-hood technology. Hence, in practice, ASP solvers are

⋆ Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

embedded in encompassing software environments and thus have to interact with them in an easy way. Again, such interactions are not accounted for by traditional ASP solvers and once more left to the user.

This shortcoming is addressed by multi-shot solvers like *clingo* 4 that embrace the dynamicity of nonmonotonic reasoning by allowing a reactive procedure to loop on solving while acquiring changes in the problem specification. Given that this is accomplished by complementing the declarative approach of ASP with procedural means, like Python or Lua, one also gets a handle on communication with an environment. In what follows, we want to illustrate these aspects by providing a hands-on introduction to multi-shot solving with *clingo* 4 through modeling the popular board game of *Ricochet Robots*. Our particular focus lies on capturing the underlying round playing through the procedural-declarative interplay offered by the Python-ASP integration of *clingo* 4.

Ricochet Robots is a board game for multiple players designed by Alex Randolph.[4] A board consists of 16×16 fields arranged in a grid structure having barriers between various neighboring fields (see Figure 1 and 2). Four differently colored robots roam across the board along either horizontally or vertically accessible fields, respectively. In principle, each robot can thus move in four directions. A robot cannot stop its move until it either hits a barrier or another robot. The goal is to place a designated robot on a target location with a shortest sequence of moves. Often this involves moving several robots to establish temporary barriers. In fact, the game is played in rounds. At each round, a chip with a colored symbol indicating the target location is drawn. Then, the specific goal is to move the robot with the same color on this location. The player who reaches the goal with the fewest number of robot moves wins the chip. The next round is then played from the end configuration of the previous round. At the end, the player with most chips wins the game.

Ricochet Robots has been studied from the viewpoint of human problem solving [5] and analyzed from a theoretical perspective [6–8]. Moreover, it has a large community providing various resources on the web. Among them, there is a collection of fifty-six extensions of the game.[5] We also studied alternative ASP encodings of the game in [9], and used them to compare various ASP solving techniques. More disparate encodings resulted from the ASP competition in 2013, where *Ricochet Robots* was included in the modeling track. ASP encodings and instances of *Ricochet Robots* are available at [10].

2 Multi-shot solving with *clingo* 4

clingo 4 offers high-level constructs for realizing complex reasoning processes that tolerate evolving problem specifications, either because data or constraints are added, deleted, or replaced. 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. As detailed in [11], *clingo* 4 complements ASP's declarative input language by control capacities expressed via the (embedded) scripting languages Lua and Python. On the declarative

[4] `http://en.wikipedia.org/wiki/Ricochet_Robot`

[5] `http://www.boardgamegeek.com/boardgame/51/ricochet-robots`

side, *clingo* 4 offers a new directive `#program` that allows for structuring logic programs into named and parametrizable 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`.

While [11] details the partition and composition of logic programs as well as the use of Python as an embedded scripting language, we focus here on the usage of externally defined atoms along with the *clingo* 4 Python library. Hence, we refer the interested reader to [11] for more details on `#program` and `#script` directives; the semantical underpinnings of program composition in terms of module theory are given in [12]. Here, it is just important to note that `base` is a dedicated subprogram that gathers all rules not preceded by a `#program` directive. Since we do not use any `#program` directives, all rules belong to the `base` program.

As detailed in the following, the `#external` directive of *clingo* 4 allows for a flexible handling of yet undefined atoms. Moreover, the (external) manipulation of their truth values provides an easy mechanism to activate or deactivate ground rules on demand. This allows for continuously assembling ground rules evolving at different stages of a reasoning process. To be more precise, `#external` directives declare atoms that may still be defined by rules added later on. As detailed in [11], in terms of modules, such atoms correspond to inputs, which 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 '`p(X,Y) :- q(X,Z), r(Z,Y).`' during grounding. However, the head atoms of the resulting ground instances are merely collected as inputs, whereas the ground rules as such are discarded.

We define a (non-ground) logic program P' as *extensible*, if it contains some (non-ground) *external declaration* of the form

$$\texttt{\#external}\ a : B \tag{1}$$

where a is an atom and B a rule body. For grounding an external declaration as in (1), it is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a distinguished ground atom marking rules from `#external` declarations. Formally, given an extensible programs P', we define the collection D of rules corresponding to `#external` declarations as follows.

$$D = \{a \leftarrow B, \varepsilon \mid (\texttt{\#external}\ a : B) \in P'\}$$

With it, the ground instantiation of the extensible logic program P' is defined as the ground logic program P associated with the set E of ground atoms, where[6]

$$P = \{r \in \mathit{grd}(P' \cup (D \cup \{\{\varepsilon\} \leftarrow\})) \setminus \{\{\varepsilon\} \leftarrow\} \mid \varepsilon \notin B(r)\} \tag{2}$$

$$E = \{h(r) \mid r \in \mathit{grd}(P' \cup (D \cup \{\{\varepsilon\} \leftarrow\})), \varepsilon \in B(r)\} \tag{3}$$

[6] We use $h(r)$ and $B(r)$ to denote the head and body of a rule r, respectively, and $\mathit{grd}(P)$ to denote the set of all ground instances of rules in P.

For simplicity, we refer to P and E as a *logic program with externals*, and drop the reference to P' whenever clear from the context. Note that $\{\varepsilon\} \leftarrow$ is added above to cope with $grd(P' \cup (D \cup \{\{\varepsilon\} \leftarrow\}))$, understood as the outcome of grounding (with simplifications).

As an example, consider the following extensible program, R':

```
#external e(X) : f(X), X < 2.
f(1..2).
a(X) :- e(X), f(X).
b(X) :- not e(X), f(X).
```

Grounding R' yields the below program R with externals $F = \{$`e(1)`$\}$.

```
f(1). f(2).
a(1) :- e(1).
b(1) :- not e(1).
b(2).
```

Note how externals influence the result of grounding. While `e(1)` remains untouched, the atom `e(2)` is set to false, followed by cascading simplifications.

For capturing the stable models of such logic programs with externals, we need the following definitions. A (partial) assignment i over a set $A \subseteq \mathcal{A}$ of atoms is a function: $i : A \rightarrow \{t, f, u\}$, where $\mathcal{A}$ is the set of given atoms. With this, we define $A^t = \{a \in A \mid i(a) = t\}$, $A^f = \{a \in A \mid i(a) = f\}$, and $A^u = \{a \in A \mid i(a) = u\}$. In what follows, we represent partial assignments either by $\langle A^t, A^f \rangle$ or $\langle A^t, A^u \rangle$ by leaving the respective default value implicit.

Given a program P with externals E, we define the set $I = E \setminus H(P)$ as input atoms of P.[7] That is, input atoms are externals that are not overridden by rules in P. Given a partial assignment $\langle I^t, I^u \rangle$ over I, we define $P_{\langle I^t, I^u \rangle} = P \cup (\{a \leftarrow \mid a \in I^t\} \cup \{\{a\} \leftarrow \mid a \in I^u\})$ to capture the extension of P with respect to an (external) truth assignment to the input I. In addition, *clingo* considers another partial assignment $\langle A^t, A^f \rangle$ over $A \subseteq \mathcal{A}$ for filtering stable models, and refers to them as assumptions.[8] Then, X is a stable model of a program P with externals E filtered by $\langle A^t, A^f \rangle$, if X is a stable model of $P_{\langle I^t, I^u \rangle}$ such that $A^t \subseteq X$ and $A^f \cap X = \emptyset$. This amounts to a semantical characterization of one-shot solving of programs with externals in *clingo* 4.

Note the difference among input atoms and (filtering) assumptions. While a true input atom amounts to a fact, a true assumption acts as an integrity constraint. Also, undefined input atoms are regarded as false, while undefined assumptions remain neutral. Finally, at the solver level, input atoms are a transient part of the representation, while assumptions only affect the assignment of a single search process.

For capturing multi-shot solving, we must account for sequences of system states, involving information about the programs kept within the grounder and the solver. To this end, we define a simple operational semantics based on system states and appropriate operations. A *clingo state* as a triple $\langle Q, P, I \rangle$ where

- Q is a (non-ground) logic program,

[7] We use $H(P) = \{h(r) \mid r \in P\}$ to denote all head atoms in P.

[8] In *clingo*, or more precisely in *clasp*, such assumptions are the principal parameter to the underlying `solve` function (see below). The term assumption traces back to [13, 14].

- P is a ground logic program,
- I is a set of input atoms along with an implicit partial assignment $\langle I^t, I^u \rangle$ over I.

Such states can be modified by the following operations.

- $\mathit{create}() : \ \mapsto \langle \emptyset, \emptyset, \emptyset \rangle$
- $\mathit{add}(R) : \langle Q, P, I \rangle \mapsto \langle Q \cup R, P, I \rangle$ where R is a (non-ground) logic program
- $\mathit{ground} : \langle Q, P_1, I_1 \rangle \mapsto \langle \emptyset, P_2, I_2 \rangle$ where[9]
  - $(P, E) = \mathit{grd}_{P_1, I_1}(Q)$
  - $P_2 = P_1 \cup P$
  - $I_2 = (I_1 \cup E) \setminus H(P_2)$
    - $I_2^t = \{a \in I_2 \mid I_1(a) = t\}$
    - $I_2^u = \{a \in I_2 \mid I_1(a) = u\}$
- $\mathit{assignExternal}(a, t) : \langle Q, P, I_1 \rangle \mapsto \langle Q, P, I_2 \rangle$ where
  - $I_2 = I_1$
    - $I_2^t = I_1^t \cup \{a\}$ if $a \in I_1$, and $I_2^t = I_1^t$ otherwise
    - $I_2^u = I_1^u \setminus \{a\}$

  $\mathit{assignExternal}(a, u) : \langle Q, P, I_1 \rangle \mapsto \langle Q, P, I_2 \rangle$ where
  - $I_2 = I_1$
    - $I_2^t = I_1^t \setminus \{a\}$
    - $I_2^u = I_1^u \cup \{a\}$ if $a \in I_1$, and $I_2^u = I_1^u$ otherwise

  $\mathit{assignExternal}(a, f) : \langle Q, P, I_1 \rangle \mapsto \langle Q, P, I_2 \rangle$ where
  - $I_2 = I_1$
    - $I_2^t = I_1^t \setminus \{a\}$
    - $I_2^u = I_1^u \setminus \{a\}$
- $\mathit{releaseExternal}(a) : \langle Q, P_1, I_1 \rangle \mapsto \langle Q, P_2, I_2 \rangle$ where
  - $P_2 = P_1 \cup \{a \leftarrow a, {\sim}a\}$ if $a \in I_1$, and $P_2 = P_1$ otherwise
  - $I_2 = I_1 \setminus \{a\}$
    - $I_2^t = I_1^t \setminus \{a\}$
    - $I_2^u = I_1^u \setminus \{a\}$
- $\mathit{solve}(\langle A^t, A^f \rangle) : \langle Q, P, I \rangle \mapsto \langle Q, P, I \rangle$ outputs the set $\mathcal{X}_{P,I}$ defined as

$$\{X \mid X \text{ is a stable model of } P_{\langle I^t, I^u \rangle} \text{ such that } A^t \subseteq X \text{ and } A^f \cap X = \emptyset\} \quad (4)$$

For simplicity, we dropped the condition '$I_2^f = I_2 \setminus (I_2^t \cup I_2^u)$' from all transitions of I_1 to I_2 because undefined input atoms are regarded to be false. Note also that the above semantic account abstracts from the partition and composition of logic programs, dealt with in [12, 11]. Rather it relies on a single (base) program whose addition complies with modularity (in terms of [15]).

[9] We use $\mathit{grd}_{P,I}(Q)$ to denote the (ground) logic program with externals obtained by instantiating the extensible program Q as defined in (2) and (3), respectively. We add the subscript P, I to indicate the context of the instantiation.

A central role is played by the *ground* function. First, programs like Q are always grounded in the context of P_1 and I_1 since they delineate the Herbrand base and universe. Second, one may also add new externals via E, provided they are not yet defined. The function *assignExternal* allows us to manipulate the truth values of input atoms. While their default value is false, making them undefined results in a choice. If an atom is not external, then *assignExternal* has no effect. On the contrary, *releaseExternal* removes the external status from an atom and sets it permanently to false, otherwise this function has no effect. Finally, *solve* leaves the *clingo* state intact and outputs the filtered set $\mathcal{X}_{P,I}$ of stable models of the logic program with externals comprised in the current state. This set is general enough to define all basic reasoning modes of *clingo*. On a technical note, the addition of $a \leftarrow a, {\sim}a$ does not offer any derivation for a but adds a to the head atoms, $H(P)$, so that it can neither be re-added as an external nor via a rule (since the latter would violate modularity [12]).

For illustration, reconsider the above extensible program R'. Adding and grounding R' in an initial state results in the *clingo* state[10] $\mathit{ground}(\mathit{add}(R')(\mathit{create}())) = \langle \emptyset, R, F^f \rangle$ where R and F are as given above. Applying $\mathit{solve}()$ to $\langle \emptyset, R, F^f \rangle$ leaves the state unaffected and results in a single stable model containing `b(1)`. Unlike this, the state $\mathit{assignExternal}(c,u)(\langle \emptyset, R, F^f \rangle)$ induces two models, one with `a(1)` and another with `b(1)`, while $\mathit{assignExternal}(c,t)(\langle \emptyset, R, F^f \rangle)$ yields only the one with `a(1)`.

From the viewpoint of operational semantics, the multi-shot solving process of a *clingo* object can be associated with the sequence of executed *clingo*-specific operations $\langle o_k \rangle_{k \in K}$ which in turn induce a sequence $\langle Q_k, P_k, I_k \rangle_{k \in K}$ of *clingo* states such that

1. $o_0 = \mathit{create}()$ and $o_k \neq \mathit{create}()$ for $k > 0$
2. $(Q_0, P_0, I_0) = \mathit{create}()$
3. $\langle Q_k, P_k, I_k \rangle = o_k(\langle Q_{k-1}, P_{k-1}, I_{k-1} \rangle)$ for $k > 0$

For capturing the result of the multi-shot solving process in terms of stable models, we consider the sequence of sets of stable models obtained at each solving step. More precisely, given a sequence of *clingo* operations and states as above, the multi-shot solving process can be associated with the sequence $\langle \mathcal{X}_{P_j, I_j} \rangle_{j \in K, o_j = \mathit{solve}(\langle A^t_j, A^f_j \rangle)}$ of sets of stable models defined in (4).

All of the above state operations have almost literal counterparts in *clingo*'s Python (and Lua) module, namely `__init__` of *clingo*'s `Control` class, `add`, `ground`, `assign_external`, `release_external`, and `solve`.[11] However, as mentioned, the above semantic account abstracts from the partition and composition of logic programs. In fact, `add` as well as `ground` associate rules with subprograms. Moreover, subprograms are usually parametrized and thus grounded several times with different instantiations of the parameters. This is not reflected by *ground* (where Q is emptied). Also, our account disregards module composition, which is enforced by *clingo* (cf. [12]).[12]

[10] We use the informal notation F^f to indicate that the members of F are false.

[11] For a complete listing of functions and classes available in the `gringo` module, see `http://potassco.sourceforge.net/gringo.html`

[12] Among others, this prevents redefining ground atoms.

Finally, it is worth mentioning that several *clingo* objects can be created and run in a multi-threaded yet independent fashion.

3 Encoding *Ricochet Robots*

The following encoding and fact formats follow the ones given in [9],[13] except that we use below the input language of *clingo* 4 that includes the ASP language standard *ASP-Core-2* [18].

An authentic board configuration of *Ricochet Robots* is shown in Figure 1 and represented as facts in Listing 1.1. The dimension of the board is fixed to 16 in Line 1. As put

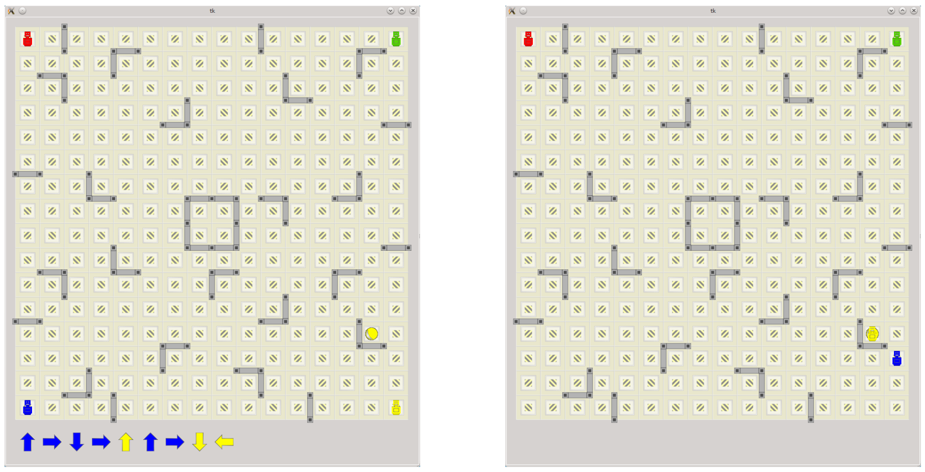

Fig. 1. Visualization of solving `goal(13)` from initially cornered robots

forward in [9], barriers are indicated by atoms with predicate `barrier`/4. The first two arguments give the field position and the last two the orientation of the barrier, which is mostly east (1,0) or south (0,1).[14] For instance, the atom `barrier(2,1,1,0)` in Line 3 represents the vertical wall between the fields (2,1) and (3,1), and `barrier(5,1,0,1)` stands for the horizontal wall separating (5,1) from (5,2).

Listing 1.1. The Board (`board.lp`)

```
1  dim(1..16).

3  barrier( 2, 1, 1,0).  barrier(13,11, 1,0).  barrier( 9, 7,0, 1).
4  barrier(10, 1, 1,0).  barrier(11,12, 1,0).  barrier(11, 7,0, 1).
5  barrier( 4, 2, 1,0).  barrier(14,13, 1,0).  barrier(14, 7,0, 1).
6  barrier(14, 2, 1,0).  barrier( 6,14, 1,0).  barrier(16, 9,0, 1).
```

[13] The encodings in [9] rely on the input language of *clingo* 3 [16, 17].

[14] Symmetric barriers are handled by predicate `stop`/4 in Line 4 and 5 of Listing 1.3.

```
barrier( 2, 3, 1,0).  barrier( 3,15, 1,0).  barrier( 2,10,0, 1).
barrier(11, 3, 1,0).  barrier(10,15, 1,0).  barrier( 5,10,0, 1).
barrier( 7, 4, 1,0).  barrier( 4,16, 1,0).  barrier( 8,10,0,-1).
barrier( 3, 7, 1,0).  barrier(12,16, 1,0).  barrier( 9,10,0,-1).
barrier(14, 7, 1,0).  barrier( 5, 1,0, 1).  barrier( 9,10,0, 1).
barrier( 7, 8, 1,0).  barrier(15, 1,0, 1).  barrier(14,10,0, 1).
barrier(10, 8,-1,0).  barrier( 2, 2,0, 1).  barrier( 1,12,0, 1).
barrier(11, 8, 1,0).  barrier(12, 3,0, 1).  barrier(11,12,0, 1).
barrier( 7, 9, 1,0).  barrier( 7, 4,0, 1).  barrier( 7,13,0, 1).
barrier(10, 9,-1,0).  barrier(16, 4,0, 1).  barrier(15,13,0, 1).
barrier( 4,10, 1,0).  barrier( 1, 6,0, 1).  barrier(10,14,0, 1).
barrier( 2,11, 1,0).  barrier( 4, 7,0, 1).  barrier( 3,15,0, 1).
barrier( 8,11, 1,0).  barrier( 8, 7,0, 1).
```

Listing 1.2 gives the sixteen possible target locations printed on the game's carton board (cf. Line 3 to 18). Each robot has four possible target locations, expressed by the ternary predicate `target`. Such a target is put in place via the unary predicate `goal` that associates a number with each location. The external declaration in Line 1 paves the way for fixing the target location from outside the solving process. For instance, setting `goal(13)` to true makes position `(15,13)` a target location for the `yellow` robot.

Listing 1.2. Robots and targets (`targets.lp`)

```
 1  #external goal(1..16).

 3  target(red,    5, 2)  :- goal(1).  % red moon
 4  target(red,   15, 2)  :- goal(2).  % red triangle
 5  target(green,  2, 3)  :- goal(3).  % green triangle
 6  target(blue,  12, 3)  :- goal(4).  % blue star
 7  target(yellow, 7, 4)  :- goal(5).  % yellow star
 8  target(blue,   4, 7)  :- goal(6).  % blue saturn
 9  target(green, 14, 7)  :- goal(7).  % green moon
10  target(yellow,11, 8)  :- goal(8).  % yellow saturn
11  target(yellow, 5,10)  :- goal(9).  % yellow moon
12  target(green,  2,11)  :- goal(10). % green star
13  target(red,   14,11)  :- goal(11). % red star
14  target(green, 11,12)  :- goal(12). % green saturn
15  target(yellow,15,13)  :- goal(13). % yellow star
16  target(blue,   7,14)  :- goal(14). % blue star
17  target(red,    3,15)  :- goal(15). % red saturn
18  target(blue,  10,15)  :- goal(16). % blue moon

20  robot(red;green;blue;yellow).
21  #external pos((red;green;blue;yellow),1..16,1..16).
```

Similarly, the initial robot positions can be set externally, as declared in Line 21. That is, each robot can be put at 256 different locations. On the left hand side of Figure 1, we cornered all robots by setting `pos(red,1,1)`, `pos(blue,1,16)`, `pos(green,16,1)`, and `pos(yellow,16,16)` to true.

Finally, the encoding in Listing 1.3 follows the plain encoding of ricocheting robots given in [9, Listing 2], yet upgraded to the input language of *clingo* 4.

Listing 1.3. Simple encoding for *Ricochet Robots* (`ricochet.lp`)

```
 1  time(1..horizon).
 2  dir(-1,0;1,0;0,-1;0,1).

 4  stop( DX, DY,X,   Y   ) :- barrier(X,Y,DX,DY).
 5  stop(-DX,-DY,X+DX,Y+DY) :- stop(DX,DY,X,Y).

 7  pos(R,X,Y,0) :- pos(R,X,Y).

 9  1 { move(R,DX,DY,T) : robot(R), dir(DX,DY) } 1 :- time(T).
10  move(R,T) :- move(R,_,_,T).

12  halt(DX,DY,X-DX,Y-DY,T) :- pos(_,X,Y,T), dir(DX,DY), dim(X-DX;Y-DY),
13                             not stop(-DX,-DY,X,Y), T < horizon.

15  goto(R,DX,DY,X,Y,T) :- pos(R,X,Y,T), dir(DX,DY), T < horizon.
16  goto(R,DX,DY,X+DX,Y+DY,T) :- goto(R,DX,DY,X,Y,T), dim(X+DX;Y+DY),
17                           not stop(DX,DY,X,Y), not halt(DX,DY,X,Y,T).

19  pos(R,X,Y,T) :- move(R,DX,DY,T), goto(R,DX,DY,X,Y,T-1),
20                  not goto(R,DX,DY,X+DX,Y+DY,T-1).
21  pos(R,X,Y,T) :- pos(R,X,Y,T-1), time(T), not move(R,T).

23  :- target(R,X,Y), not pos(R,X,Y,horizon).

25  #show move/4.
```

Following the description in [9], the first lines in Listing 1.3 furnish domain definitions, fixing the sequence of time steps (`time`/1)[15] and two-dimensional representations of the four possible directions (`dir`/2). The constant `horizon` is expected to be provided via *clingo* option `-c` (eg. '`-c horizon=20`'). Predicate `stop`/4 is the symmetric version of `barrier`/4 from above and identifies all blocked field transitions. The initial robot positions are fixed in Line 7 (in view of external input).

At each time step, some robot is moved in a direction (cf. Line 9). Such a `move` can be regarded as the composition of successive field transitions, captured by predicate `goto`/6 (in Line 15–17). To this end, predicate `halt`/5 provides temporary barriers due to robots' positions before the `move`. To be more precise, a robot moving in direction `(DX,DY)` must halt at field `(X-DX,Y-DY)` when some (other) robot is located at `(X,Y)`, and an instance of `halt(DX,DY,X-DX,Y-DY,T)` may provide information relevant to the `move` at step `T+1` if there is no barrier between `(X-DX,Y-DY)` and `(X,Y)`. Given this, the definition of `goto`/6 starts at a robot's position (in Line 15) and continues in direction `(DX,DY)` (in Line 16–17) unless a barrier, a robot, or the board's border is encountered. As this definition tolerates board traversals of length zero, `goto`/6 is guaranteed to yield a successor position for any `move` of a robot `R` in direction `(DX,DY)`, so that the rule in Line 19–20 captures the effect of `move(R,DX,DY,T)`. Moreover,

[15] The initial time point `0` is handled explicitly.

the frame axiom in Line 21 preserves the positions of unmoved robots, relying on the projection `move`/2 (cf. Line 10).

Finally, we stipulate in Line 23 that a robot `R` must be at its target position `(X,Y)` at the last time point `horizon`. Adding directive '`#show move/4.`' further allows for projecting stable models onto the extension of the `move`/4 predicate.

The encoding in Listing 1.3 allows us to decide whether a plan of length `horizon` exists. For computing a shortest plan, we may augment our decision encoding with an optimization directive. This can be accomplished by adding the part in Listing 1.4.

Listing 1.4. Encoding part for optimization (`optimization.lp`)

```
27  goon(T) :- target(R,X,Y), T = 0..horizon, not pos(R,X,Y,T).

29  :- move(R,DX,DY,T-1), time(T), not goon(T-1), not move(R,DX,DY,T).

31  #minimize{ 1,T : goon(T) }.
```

The rule in Line 27 indicates whether some goal condition is (not) established at a time point. Once the goal is established, the additional integrity constraint in Line 29 ensures that it remains satisfied by enforcing that the goal-achieving move is repeated at later steps (without altering robots' positions). Note that the `#minimize` directive in Line 31 aims at few instances of `goon`/1, corresponding to an early establishment of the goal, while further repetitions of the goal-achieving move are ignored. Our extended encoding allows for computing a shortest plan of length bounded by `horizon`. If there is no such plan, the problem can be posed again with an enlarged `horizon`. For computing a shortest plan in an unbounded fashion, we can take advantage of incremental ASP solving, as detailed in [9].[16]

Apart from the two external directives that allow us to vary initial robot and target positions, the four programs constitute an ordinary ASP formalization of a *Ricochet Robots* instance. To illustrate this, let us override the external directives by adding facts accounting for the robot and target positions on the left hand side of Figure 1. The corresponding call of *clingo* 4 is shown in Listing 1.5.[17]

Listing 1.5. One-shot solving with *clingo* 4

```
$ clingo-4 board.lp targets.lp ricochet.lp optimization.lp   \
        -c horizon=10                                        \
        <(echo "pos(red,1,1).   pos(green,16,1).   \
                pos(blue,1,16). pos(yellow,16,16). \
                goal(13)."                                   )
```

Listing 1.6. Stable model projected onto the extension of the `move`/4 predicate

```
move(blue,0,-1,1)      move(blue,1,0,2)       move(blue,0,1,3)      \
move(blue,1,0,4)       move(yellow,0,-1,5)    move(blue,0,-1,6)     \
move(blue,1,0,7)       move(yellow,0,1,8)     move(yellow,-1,0,9)   \
move(yellow,-1,0,10)
```

[16] Note that [9] uses *iclingo* [14] for incremental solving. This functionality is now part of *clingo* 4 and makes *iclingo* obsolete. See [11] for details.

[17] Note that rather than using input redirection, we also could have passed the five facts via a file.

The resulting one-shot solving process yields a(n optimal) stable model containing the extension of the `move`/4 predicate given in Listing 1.6. The `move` atoms in Line 1–4 of Listing 1.6 correspond to the plan indicated by the colored arrows at the bottom of the left hand side of Figure 1. That is, the blue robot starts by going north, east, south, and east, then the yellow one goes north, the blue one resumes and goes north and east, before finally the yellow robot goes south (bouncing off the blue one) and lands on the target by going west. This leads to the situation depicted on the right hand side of Figure 1. Note that the tenth move (in Line 4) is redundant since it merely replicates the previous one because the goal was already reached after nine steps.

4 Playing in rounds

Ricochet Robots is played in rounds. Hence, the next goal must be reached with robots placed at the positions resulting from the previous round. For example, when pursuing `goal(4)` in the next round, the robots must start from the end positions given on the right hand side of Figure 1. The resulting configuration is shown on the left hand side of Figure 2. For one-shot solving, we would re-launch *clingo* 4 from scratch as shown

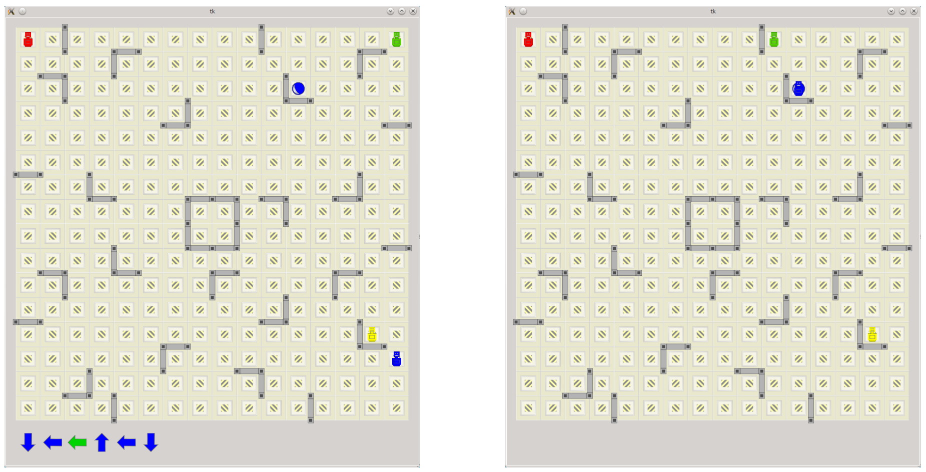
Fig. 2. Visualization of solving `goal(4)` from robot positions after having solved `goal(13)`

in Listing 1.5, yet by accounting for the new target and robot positions by replacing Line 3–5 of Listing 1.5 by the following ones.

```
3      <(echo "pos(red,1,1).   pos(green,16,1).    \
4              pos(blue,16,10). pos(yellow,15,13). \
5              goal(4)."                                  )
```

Unlike this, our multi-shot approach to playing in rounds relies upon a single[18] operational *clingo* control object that we use in a simple loop:

[18] In general, multiple such control objects can be created and made to interact via Python.

1. Create an operational control object (containing a grounder and a solver object)
2. Load and ground the programs in Listing 1.1, 1.2, 1.3, and optionally 1.4 (relative to some fixed `horizon`) within the control object
3. While there is a goal, do the following
   (a) Enforce the initial robot positions
   (b) Enforce the current goal
   (c) Solve the logic program contained in the control object

The control loop is implemented in Python and relies on *clingo*'s Python module accompanying *clingo* 4.4. This module provides grounding and solving functionalities. An analogous module is available for Lua. As mentioned in Section 2, both modules support (almost) literal counterparts to 'Create', 'Load', 'Ground', and 'Solve'. The "enforcement" of robot and target positions is more complex, as it involves changing the truth values of externally controlled atoms (mimicking the insertion and deletion of atoms, respectively).

The resulting Python program is given in Listing 1.7. This program as well as its Lua counterpart are available at [10]. Line 1 shows how to import the `gringo` module.[19] We are only using three classes from the module,[20] which we directly pull into the global namespace to avoid qualification with "`gringo.`" and so to keep the code compact.

Line 3–34 show the `Player` class. This class encapsulates all state information including *clingo*'s `Control` object that in turn holds the state of the underlying grounder and solver. In the `Player`'s `__init__` function (similar to a constructor in other object-oriented languages) the following member variables are initialized:

`last_positions` This variable is initialized upon construction with the starting positions of the robots. During the progression of the game, this variable holds the initial starting positions of the robots for each turn.

`last_solution` This variable holds the last solution of a search call.

`undo_external` We want to successively solve a sequence of goals. In each step, a goal has to be reached from different starting positions. This variable holds a list containing the current goal and starting positions that have to be cleared upon the next step.

`horizon` We are using a bounded encoding. This (Python) variable holds the maximum number of moves to find a solution for a given step.

`ctl` This variable holds the actual object providing an interface to the grounder and solver. It holds all state information necessary for multi-shot solving along with heuristic information gathered during solving.

As shown in Line 4–13, the constructor takes the `horizon`, initial robot `positions`, and the `files` containing the various logic programs. *clingo*'s `Control` object is created in Line 9–10 by passing the option `-c` to replace the logic program constant `horizon` by the value of the Python variable `horizon` during grounding. Finally, the

[19] For historical reasons, it is called `gringo` in *clingo* 4.4 but it will be renamed to `clingo` with the next release.

[20] For a complete listing of functions and classes available in the `gringo` module, see `http://potassco.sourceforge.net/gringo.html`

Listing 1.7. The Ricochet Robot Player (`ricochet.py`)

```python
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(
            [’-c’, ’horizon={0}’.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and
                    len(atom.args()) == 4 and
                    atom.args()[3] == self.horizon):
                self.last_positions.append(
                    Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]),
             Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]),
             Fun("goal", [4]),
             Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

constructor loads all `files` and grounds the entire logic program in Line 11–13. Recall from Section 2 that all rules outside the scope of `#program` directives belong to the `base` program. Note also that this is the only time grounding happens because the encoding is bounded. All following solving steps are configured exclusively via manipulating external atoms.

The `solve` method in Line 15–24 starts with initializing the search for the solution to the new `goal`. To this end, it first undos in Line 16–17 the previous goal and starting positions stored in `undo_external` by assigning `False` to the respective atoms. In the following lines 19 to 21, the next step is initialized by assigning `True` to the current `goal` along with the last robot positions; these are also stored in `undo_external` so that they can be taken back afterwards. Finally, the `solve` method calls *clingo*'s `ctl.solve` to initiate the search. The result is captured in variable `last_solution`. Note that the call to `ctl.solve` takes `ctl.on_model` as (keyword) argument, which is called whenever a model is found. In other words, `on_model` acts as a callback for intercepting models. Finally, variable `last_solution` is returned at the end of the method.

The last function of the `Player` class is the `on_model` callback. As mentioned, it intercepts the (final) `model`s computed by the solver, which can then be inspected via the functions of the `Model` class. At first, it stores the shown atoms in variable `last_solution` in Line 27.[21] The remainder of the `on_model` callback extracts the final robot positions from the stable model. For that, it loops in Line 29–34 over the full set of atoms in the `model` and checks whether their signatures match. That is, if an atom is formed from predicate `pos/4` and its fourth argument equals the `horizon`, then it is appended to the list of `last_positions` after stripping its time step from its arguments.

As an example, consider `pos(yellow,15,13,20)`, say the final position of the yellow robot on the right hand side of Figure 1 at an `horizon` of 20. This leads to the addition of `pos(yellow,15,13)` to the `last_positions`. Note that `pos(yellow,15,13)` is declared an external atom in Line 21 of Listing 1.2. For playing the next round, we can thus make it `True` in Line 20 of Listing 1.7. And when solving, the rule in Line 7 of Listing 1.3 allows us to derive `pos(yellow,15,13,0)` and makes it the new starting position of the yellow robot, as shown on the left hand side of Figure 2.

Line 36–44 show the code for configuring the player. They set the search `horizon`, the `encodings` to solve with, and the initial `positions` in form of `gringo` terms. Furthermore, we fix a `sequence` of goals in Line 42–44. In a more realistic setting, either some user interaction or a random sequence might be generated to emulate arbitrary draws.

Listing 1.8. Multi-shot solving with *clingo* 4's Python module

```
$ python ricochet.py
[move(red,0,1,1), move(red,1,0,2), move(red,0,1,3), ...]
[move(blue,0,-1,1), move(blue,1,0,2), move(blue,0,1,3), ...]
[move(green,0,1,1), move(green,1,0,2), move(green,1,0,3), ...]
```

[21] In view of '`#show move/4.`' in Listing 1.3, this only involves instances of `move/4`, while all true atoms are included via the argument `Model.ATOMS` in Line 29.

Finally, Line 46–48 implement the search for sequences of moves that solve the configuration given above. For each `goal` in the `sequence`, a solution is plainly printed, as engaged in Line 48. The three lists in Listing 1.8 represent solutions to the three goals in Line 42–44. The *clingo* library does not foresee any output, which must thus be handled by the scripting language. Note also that the first list represents an alternative solution to the one given in Listing 1.6.

5 Discussion

Multi-shot ASP solving is about successive yet operational grounding and solving of changing logic programs due to the addition, deletion, or replacement of facts or rules. Special cases include incremental, reactive, and window-based solving. For addressing such complex reasoning processes, *clingo* 4 complements ASP's declarative input language by control capacities expressed via the scripting languages Lua and Python. We elaborated upon *clingo*'s high-level constructs supporting multi-shot solving in several ways. First, we provided an operational semantics based on the concepts of *clingo* states and associated operations. These operations reflect the major functionalities offered by *clingo*'s Lua and Python library. A particular focus lay on the instantiation of extensible non-ground programs leading to ground programs with externals. Such externals are the primary means for changing problem specifications. Second, we provided a hands-on introduction to multi-shot solving with *clingo* 4 by modeling the popular board game of *Ricochet Robots*. In particular, we showed how *clingo*'s Python library allows for modeling turn playing by manipulating externals. Finally, we hope that our ASP-based implementation helps Gerd to win more often at *Ricochet Robots*.

Acknowledgments. This work was partially funded by German Science Foundation (DFG) under grant SCHA 550/9-1.

References

1. Bobrow, D., ed.: Special issue on nonmonotonic logic. Volume 13. Artificial Intelligence (1980)
2. Brewka, G.: Nonmonotonic Reasoning: From Theoretical Foundation to Efficient Computation. Dissertation, Universität Hamburg (1989) Revised Version appeared as: Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (1990)
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
4. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54**(12) (2011) 92–103
5. Butko, N., Lehmann, K., Ramenzoni, V.: Ricochet Robots — a case study for human complex problem solving. In: Proceedings of the Annual Santa Fe Institute Summer School on Complex Systems (CSSS'05). (2005)
6. Engels, B., Kamphans, T.: On the complexity of Randolph's robot game. Research Report 005, Institut für Informatik, Universität Bonn (2005)
7. Engels, B., Kamphans, T.: Randolph's robot game is NP-hard! Electronic Notes in Discrete Mathematics **25** (2006) 49–53

8. Engels, B., Kamphans, T.: Randolph's robot game is NP-complete! In: Proceedings of the Twenty-second European Workshop on Computational Geometry (EWCG'06). (2006) 157–160
9. Gebser, M., Jost, H., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T., Schneider, M.: Ricochet robots: A transverse ASP benchmark. In Cabalar, P., Son, T., eds.: Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13). Volume 8148 of Lecture Notes in Artificial Intelligence. Springer-Verlag (2013) 348–360
10. `http://potassco.sourceforge.net/apps.html`
11. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo* = ASP + control: Preliminary report. In Leuschel, M., Schrijvers, T., eds.: Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14). Theory and Practice of Logic Programming, Online Supplement (2014) see also arXiv:1405.3694v1
12. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo* = ASP + control: Extended report. (2014) `http://www.cs.uni-potsdam.de/wv/pdfformat/gekakasc14a.pdf`
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4) (2003)
14. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In Garcia de la Banda, M., Pontelli, E., eds.: Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Volume 5366 of Lecture Notes in Computer Science. Springer-Verlag (2008) 190–205
15. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06). IOS Press (2006) 412–416
16. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. In Delgrande, J., Faber, W., eds.: Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Volume 6645 of Lecture Notes in Artificial Intelligence. Springer-Verlag (2011) 345–351
17. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. (2010) `http://sourceforge.net/projects/potassco/files/potassco_guide/2010-10-04/guide.pdf`
18. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., Schaub, T.: ASP-Core-2: Input language format. (2012) `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf`