Ricochet Robots: A transverse ASP benchmark

Martin Gebser, Holger Jost, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, Torsten Schaub, and Marius Schneider

Universität Potsdam

Abstract. A distinguishing feature of Answer Set Programming is its versatility. In addition to satisfiability testing, it offers various forms of model enumeration, intersection or unioning, as well as optimization. Moreover, there is an increasing interest in incremental and reactive solving due to their applicability to dynamic domains. However, so far no comparative studies have been conducted, contrasting the respective modeling capacities and their computational impact. To assess the variety of different forms of ASP solving, we propose Alex Randolph's board game *Ricochet Robots* as a transverse benchmark problem that allows us to compare various approaches in a uniform setting. To begin with, we consider alternative ways of encoding ASP planning problems and discuss the underlying modeling techniques. In turn, we conduct an empirical analysis contrasting traditional solving, optimization, incremental, and reactive approaches. In addition, we study the impact of some boosting techniques in the realm of our case study.

1 Introduction

A distinguishing feature of Answer Set Programming (ASP; [1]) is its versatility. Its modeling language and solving technology support various forms of (Boolean) constraint solving that are otherwise restricted to dedicated paradigms. For example, optimization is not supported by standard Satisfiability solvers, and one has to resort to Maximum Satisfiability solvers to obtain this functionality (cf. [2]). Unlike this, ASP solvers offer, in addition to satisfiability testing, various forms of model enumeration, intersection or unioning, as well as (multi-criteria) optimization. Moreover, there is an increasing interest in incremental and reactive solving due to their applicability to dynamic domains, such as assisted living and cognitive robotics. However, so far no comparative studies have been conducted, contrasting the respective modeling capacities and their computational impact.

To assess the variety of different forms of ASP solving, we propose the popular board game *Ricochet Robots* as a transverse benchmark problem that allows us to compare various approaches in a uniform setting. *Ricochet Robots* is a board game for multiple players designed by Alex Randolph.[1] A board consists of 16×16 fields arranged in a grid structure having barriers between various neighboring fields. 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 hits either a barrier or another robot. Finally, the goal is to

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

place a particular robot on a target location with a shortest sequence of moves. Often this involves moving several robots to establish temporary barriers. For illustration, consider the reduced board in Figure 1.[2] The red robot can be moved onto the red icon in four steps: down, right, up, and left. The game box offers 96 distinct boards, each of which has sixteen (plus one special) target locations. The overall game is won by the player who wins the majority of individual rounds. (Note that the skill of human players tends to improve over the rounds because they gather knowledge about the board at hand.) *Ricochet Robots* has been studied from the viewpoint of human problem solving [3] and analyzed from a theoretical perspective [4–6]. 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.[3]

Alex Randolph's *Ricochet Robots* represent a challenging planning problem involving several actors. As such, it allows us to elaborate upon various aspects of ASP. We begin by addressing the corresponding decision problem of whether there is a plan of length smaller or equal than a given horizon. Starting from a plain encoding following traditional ASP planning (cf. [7]), we elaborate upon an alternative encoding featuring several advanced modeling techniques. We further adapt encodings for applying optimization, incremental, and reactive ASP solving techniques. While optimization allows for computing a shortest plan smaller or equal than a given horizon, the incremental and reactive variants do not impose such an upper bound. We use the devised encodings to conduct a comparative empirical analysis addressing the following questions. How do modeling techniques affect the grounding and solving performance of ASP systems? Second, to what extent can algorithm configuration as well as multi-threaded solving speed up the search for an arbitrary or a shortest plan, respectively? Furthermore, how does (bounded) optimization with standard solving techniques compare to (unbounded) optimization using incremental solving? Finally, does reactive ASP solving benefit from proceeding over a series of rounds, rather than tackling them independently?

Last but not least, we provide the visualization tool *robotviz* that, given a stable model, allows us to inspect the corresponding *Ricochet Robots* board on the screen and to interactively trace a plan described by the stable model.

2 Encoding *Ricochet Robots*

In what follows, we present our fact format and two alternative encodings in the input language of the ASP grounder *gringo* 3 [8, 9].

2.1 Fact format

For illustration, consider the reduced board of 8×8 fields in Figure 1 (corresponding to a quarter of an authentic board). Its representation in terms of facts is given in Listing 1. The board size is fixed via the constant `dimension`; its origin (1,1) is in the upper left corner. Barriers are indicated by atoms with predicate `barrier`/4. The first two arguments

[2] The enclosed yellow robot is an artifact, given that we took a quarter of an authentic board.

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

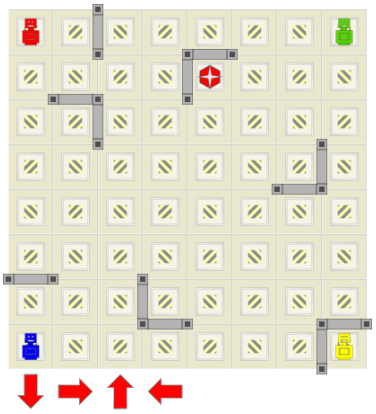

Fig. 1. Visualization of solving `board8.lp` through *robotviz*

Listing 1. Example problem instance (`board8.lp`)

```
#const dimension=8.

barrier(2,1,1,0).           barrier(5,1,0,1).
barrier(2,3,1,0).           barrier(2,2,0,1).
barrier(3,7,1,0).           barrier(7,4,0,1).
barrier(4,2,1,0).           barrier(1,6,0,1).
barrier(7,4,1,0).           barrier(4,7,0,1).
barrier(7,8,1,0).           barrier(8,7,0,1).

position(red,1,1).          position(yellow,dimension,dimension).
position(blue,1,dimension). position(green,dimension,1).
robot(R) :- position(R,_,_).                target(red,5,2).
```

give the field position and the last two the orientation of the barrier, which is either east (1,0) or south (0,1). For instance, `barrier(2,1,1,0)` 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). As specified by atoms with predicate `position`/3, the four robots start from board corners. Since each robot has (exactly one) initial position, the projection `robot`/1 captures available robots. Finally, `target(red,5,2)` expresses that the goal is to move the red robot on the red icon, as displayed in Figure 1.

2.2 Plain encoding

In Listing 2, we provide an encoding following common practice in ASP planning [7]. That is, sequences of actions are guessed via choice rules (in Line 10), and the respective successor states are derived via direct effect and frame axioms (in Line 20–22).

In more detail, the first three lines in Listing 2 furnish domain definitions, fixing the sequence of time steps (`time`/1),[4] the coordinates of board fields (`dim`/1), and two-dimensional representations of the four possible directions (`dir`/2). The constant `horizon`, used to define `time`/1, is expected to be provided via *gringo*'s command line option `-c` (e.g. '`-c horizon=10`'). Predicate `stop`/4, which is the symmetric version of

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

Listing 2. Plain encoding of ricocheting robots

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

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

8  position(R,X,Y,0) :- position(R,X,Y).

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

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

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

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

24 :- target(R,X,Y), not position(R,X,Y,horizon).
```

Listing 3. Encoding part for optimization

```
24 goon(T) :- target(R,X,Y), T := 0..horizon, not position(R,X,Y,T).
25 :- goon(horizon).

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

29 #minimize{ goon(_) }.
```

`barrier`/4 from a problem instance, identifies all blocked field transitions. The initial robot positions are fixed in Line 8.

At each time step, some robot is moved in a direction (cf. Line 10). Such a `move` can be regarded as the composition of successive field transitions, captured by `goto`/6 (in Line 16–18). 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 16) and continues in direction `(DX,DY)` (in Line 17–18) unless a barrier, a robot, or the board's border is encountered. As this definition tolerates board traversals of length zero, `goto`/6 yields a successor position for any `move` of a robot `R` in direction `(DX,DY)`, so that the rule in Line 20–21 captures the effect of `move(R,DX,DY,T)`. Moreover, the frame axiom in Line 22 preserves the positions of unmoved robots, relying on the projection `move`/2 (cf. Line 11).

Finally, we stipulate in Line 24 that a robot `R` must be at its target position `(X,Y)` at the last time point `horizon`. Adding directives '**`#hide. #show`** `move/4.`' further allows for projecting stable models onto the extension of the `move`/4 predicate.

The encoding in Listing 2 allows us to decide whether a plan of length `horizon` exists. For computing a shortest plan, we have two options resting on extended ASP

systems. The first alternative is to augment our decision encoding with an optimization directive. This can be accomplished by replacing the integrity constraint in Line 24 in Listing 2 by the encoding part in Listing 3. The new rule in Line 24 indicates whether some goal condition is (not) established at a time point, and compliance with target position(s) at the last time point `horizon` is checked by the integrity constraint in Line 25. Once the goal is established, the additional integrity constraint in Line 27 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 29 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. This allows us to successively explore all bounds from 1 on until a plan is found. An incremental ASP encoding consists of three types of rules: static, cumulative, and volatile ones. Static rules, indicated by **`#base`**, describe step-independent knowledge. Rules that are accumulated over steps are declared with **`#cumulative`** (along with a constant standing for the step number), whereas the ones stated as **`#volatile`** are specific to each time step and discarded upon incrementing the step counter.[5] An incremental ASP encoding is obtained from the one in Listing 2 as follows. First, delete Line 1 and all occurrences of `time(T)`. Second, insert '**`#base.`**' in Line 1, '**`#cumulative t.`**' in Line 9, and '**`#volatile t.`**' in Line 23. Third, replace all occurrences of variable `T` by the constant `t` or by `t-1` in Line 13–18, respectively. Finally, replace `horizon` in Line 24 by constant `t`. In doing so, we declare the rules in Line 2–8 to be static, those in Line 10–22 to be cumulative, and Line 24 to be volatile. Given this, an incremental ASP system first grounds the static part, and then it successively grounds (replacing constant `t` by the current step number) and solves the cumulative and volatile rules, incrementing the step counter until the first plan is found.

The incremental variant of Listing 2 can also be used in a reactive setting [12]. In fact, the only change concerns the way we deal with consecutive target positions. For this purpose, it is sufficient to replace the static target in each problem instance, for example '`target(red,5,2).`' in Listing 1, by the following choice rule:

```
1 { target(R,X,Y) : robot(R) : dim(X;Y) } 1.
```

This rule leaves the concrete target position open. Consecutive queries are then posed to a reactive ASP system via a sequence of **`#volatile`** integrity constraints, like '`:- not target(red,5,2).`' The volatile nature of each query guarantees that it vanishes after it has been addressed. Note that, for the sake of comparability, we refrain from modifying the initial robot positions. We discuss an experiment that simulates "playing in rounds" in Section 3.

2.3 Advanced encoding

Next, we present an advanced encoding that differs from the above in two salient ways. Its basic ideas are to guess states rather than actions and to split robots' positions into

[5] For a detailed introduction to incremental ASP, the interested reader is referred to [10, 11].

Listing 4. Advanced encoding of ricocheting robots

```
1  time(1..horizon).
2  dim(1..dimension).
3  dir(-1,0;;1,0;;0,-1;;0,1). aux(-1;1).

5  stop( DX,  0,Y,X      ) :- barrier(X,Y,DX,0).
6  stop(  0, DY,X,Y      ) :- barrier(X,Y,0,DY).
7  stop(-DX,-DY,F,L+DX+DY) :- stop(DX,DY,F,L).

9  spot(R, 1,X,0) :- position(R,X,_).
10 spot(R,-1,Y,0) :- position(R,_,Y).

12 same(R,A,RR,T) :- spot(R;RR,A,L,T), R != RR, T < horizon.

14 halt(R,DX,DY,L,      T) :- spot(R,|DY|-|DX|,F,T), stop(DX,DY,F,L), T < horizon.
15 halt(R,DX,DY,L-DX-DY,T) :- same(R,|DY|-|DX|,RR,T), spot(RR,|DX|-|DY|,L,T),
16                            dir(DX,DY), dim(L-DX-DY).

18 goto(R,DX,DY,L,      T) :- spot(R,|DX|-|DY|,L,T), dir(DX,DY), T < horizon.
19 goto(R,DX,DY,L+DX+DY,T) :- goto(R,DX,DY,L,T), dim(L+DX+DY), not halt(R,DX,DY,L,T).

21 goto(R,|DX|-|DY|,L,T) :- goto(R,DX,DY,L,T).
22 halt(R,|DX|-|DY|,L,T) :- halt(R,DX,DY,L,T), dim(L-DX-DY).

24 1 { spot(R,A,L,T) : dim(L) } 1 :- robot(R), aux(A), time(T).
25 :- spot(R,A,L,T), time(T), not goto(R,A,L,T-1).

27 bump(R,A,L,T) :- spot(R,A,L,T), time(T), not spot(R,A,L,T-1).
28 bump(R,A,L,T) :- bump(R,A,L,T-1), time(T), not goon(T-1).
29 bump(R,A,  T) :- bump(R,A,_,T).
30 :- bump(R,A,L,T), dim(L+D) : aux(D), not halt(R,A,L,T-1).
31 :- time(T), not #count{ bump(_,_,_,T) } 1.
32 :- time(T), not bump(R,A,T) : robot(R) : aux(A).
33 :- bump(R,A,T-1;T), goon(T-1).

35 goon(T) :- target(R,X,_), T := 0..horizon, not spot(R, 1,X,T).
36 goon(T) :- target(R,_,Y), T := 0..horizon, not spot(R,-1,Y,T).
37 :- goon(horizon).

39 move(R,DX,DY,T) :- bump(R,|DX|-|DY|,L,T), halt(R,DX,DY,L,T-1) : dim(L+1),
40                    dir(DX,DY), 0 < DX+DY.
41 move(R,DX,DY,T) :- bump(R,|DX|-|DY|,T), not move(R,-DX,-DY,T),
42                    dir(DX,DY), DX+DY < 0.
```

horizontal and vertical coordinates. The first idea is conceptually different from standard encodings in ASP planning, like the one above, and has the advantage that states need not be constructed by effect and frame axioms. The second idea is well-known in automated planning and leads to a significant reduction in the size of ground instantiations. On the other hand, it makes the encoding more complex since robots' positions are not given directly anymore. Further modeling techniques are described on the fly.

Our advanced encoding is given in Listing 4. Apart from two auxiliary atoms, `aux(-1)` and `aux(1)`, to distinguish horizontal and vertical coordinates, Line 1–3 are as in Listing 2. Moreover, the definition of `stop`/4 is analogous, except that its format is lined up for one-dimensional movements. In particular, columns `X` and rows `Y` are transposed in Line 5 to let `stop(DX,DY,F,L)` represent that `(F,L+DX+DY)` is inaccessible from `(F,L)` (cf. Line 7), where `F` is a fixed row or column, respectively.

The one-dimensional layout is continued with predicate `spot`/4 in Line 9, 10, and 24, where atoms `spot(R,1,X,T)` and `spot(R,-1,Y,T)` provide the column `X`

and row `Y` of robot `R` at time point `T`. Note that such coordinates are guessed, rather than derived from moves, in Line 24. In order to compensate for the split positions, the rule in Line 12 defines `same(R,A,RR,T)` (for all but the last time point `horizon`) to express that distinct robots `R` and `RR` are in a common column (`A = 1`) or row (`A = -1`).

The one-dimensional counterparts `halt`/5 and `goto`/5 of corresponding predicates in Listing 2 capture the effect of moving a robot `R` in direction `(DX,DY)` at step `T+1`, possibly altering its column (`|DX|-|DY| = 1`) or row (`|DX|-|DY| = -1`). In fact, barriers and other robots `RR` sharing the coordinate of `R` on the orthogonal axis `|DY|-|DX|` block transitions in direction `(DX,DY)`. This is captured by the definition of halt coordinates for `R` in direction `(DX,DY)` in Line 14–16. In turn, starting from its coordinate `L` on axis `|DX|-|DY|` (in Line 18), a robot `R` continues in direction `(DX,DY)` (in Line 19) unless a halt coordinate or the board's border is encountered.

Given `halt`/5 and `goto`/5, the rules in Line 21 and 22 provide the abstractions `halt`/4 and `goto`/4, which summarize coordinates affected by horizontal or vertical moves by collapsing directions `(DX,DY)` to `|DX|-|DY|`. As a minor optimization, we drop coordinates at the board's border in `halt`/4 because moves may always halt there.

With predicates providing properties of the predecessor state at hand, we can now constrain successor states guessed in Line 24. First of all, the reachability of successor coordinates along axes is checked by the integrity constraint in Line 25; that is, no robot is allowed to cross barriers or other robots' positions. A new coordinate `L` for a robot `R` on axis `A` at time step `T` is indicated by deriving `bump(R,A,L,T)` in Line 27; such atoms point to moves, and the integrity constraint in Line 31 restricts their number to (at most) one per time step. Moreover, the integrity constraint in Line 30 checks that halting at a new coordinate is admissible, which is trivially the case for coordinates at the board's border. The second possibility of deriving `bump(R,A,L,T)` in Line 28, by which a goal-achieving move at `T-1` is (necessarily) repeated at time step `T`, relies on the absence of `goon(T-1)` (defined in Line 35–36) for indicating that some goal condition is not yet established. Along with the integrity constraint in Line 31, the rule in Line 28 suppresses any further move after establishing the goal, while still supplying the projection `bump`/3 (cf. Line 29). The latter is investigated by the integrity constraint in Line 32, stipulating some instance of `bump`/3 to hold at each time step.[6] Finally, the integrity constraint in Line 33 discards redundant successive moves of a robot `R` on the same axis `A` at time steps `T-1` and `T` unless the goal is established at `T-1`.

As in Listing 3, the integrity constraint in Line 37 checks compliance with target position(s) at the last time point `horizon`. Moreover, the definition of `move`/4 in Line 39–42 provides the same format for moves as obtained with the plain encoding in Listing 2, while not carrying relevant information regarding the existence of a plan. Furthermore, note that our advanced encoding can be customized to compute shortest plans, either by adding the **`#minimize`** directive in Line 29 in Listing 3 or by devising an incremental ASP encoding according to the scheme described in Section 2.2.

[6] W.l.o.g., we assume that the goal is not readily established at the initial time point 0.

2.4 Output format and visualization

The *Ricochet Robots* visualization tool *robotviz* allows for displaying the board with barriers, robots, and targets as well as for animating robot moves in a stepwise fashion. *robotviz* is written in C++ and uses *clasp*'s textual output as input. That is, stable models are simply piped into *robotviz*, where they are parsed with a simple string parser. The first stable model is used for interactive visualization. Also, depending on the input, only board and barriers can be displayed or additional robots and a series of moves. For an impression, note that Figure 1 shows a snapshot of *robotviz*. It allows us to visually observe that the yellow robot is trapped in its corner. In addition, the plan of moving the red robot down, right, up, and left is displayed below the board in terms of a sequence of arrows in moved robots' colors, and the steps can further be traced via cursor keys.

The input format is designed for multiple encodings. For this, it is sufficient that certain key atoms belong to the solver's output and are thus declared via appropriate `#show` statements. Barriers are extracted from atoms with predicate `barrier`/4. Analogously, the robots' target and starting positions as well as their moves have to be provided for the interactive step-by-step visualization. If the input lacks `position`/3 or `move`/4, only the board is displayed. Finally, the atom `dim(dimension)` must be shown to indicate the board size (otherwise it would not be visible in the solver output).

3 Experimental case studies

Our benchmark set is based on an authentic board designed by Alex Randolph of size 16×16. The initial robot positions are in the corners of the board, and the red robot must reach some target position. Given this setting, we obtain a collection of 256 benchmark instances by considering all available fields as target positions. With very few exceptions, the resulting instances are satisfiable when given enough steps, where about twelve steps are required on average. In the following, we first focus on a comparison of encodings as well as solving strategies for decision and optimization tasks. Afterwards, we extend the scope to incremental and reactive ASP systems. All our experiments were run on a Linux machine equipped with two Quad-Core Xeon E5520 2.27GHz processors and 24GB memory, limiting each run to 600 seconds wall-clock time.

3.1 Encodings and configurations

In an ASP production mode, the most important factor is a scalable encoding. In fact, our advanced encoding in Listing 4 has a clear edge on the plain one in Listing 2 as regards grounding. This can roughly be quantified by a factor of five in terms of time consumption and ground instantiation size. However, space savings due to split positions also incorporate some indirection in referring to the actual fields of robots. Hence, it is interesting to compare search performance relative to encodings in solving decision and optimization tasks. To this end, we fix the constant `horizon` to 20 steps, which (in all but two cases) is sufficient to find plans for satisfiable instances.

In what follows, we investigate the impact of encodings and settings on solver performance. To this end, we consider *clasp* (2.1.3) in its default configuration (including

		Decision problem			Optimization problem		
		Runtime	Timeout	PAR10	Runtime	Timeout	PAR10
plain	*clasp* w `berkmin`	144	25	671	334	99	2422
	clasp w `vsids`	136	37	916	299	94	2281
	clasp, manually configured	103	14	398	234	69	1689
	clasp, automatically configured	150	28	741	259	84	2031
	clasp, multi-threaded	62	4	146	173	50	1178
advanced	*clasp* w `berkmin`	207	63	1536	302	106	2537
	clasp w `vsids`	140	48	1152	315	114	2720
	clasp, manually configured	65	12	318	192	61	1478
	clasp, automatically configured	44	9	234	136	**35**	**874**
	clasp, multi-threaded	**24**	**3**	**87**	**123**	37	904

Table 1. Solving decision and optimization problems with different encodings and *clasp* settings

`--heuristic=berkmin`) and the variant with `--heuristic=vsids`, both serving as points of reference. We further contrast these two settings with the following ones:

1. the *clasp* configuration used for the ASP competition in 2013; originally obtained by manual tuning and extensive experimentation; now available via the option `--configuration=handy` in *clasp* (2.1.3),
2. an automatically generated *clasp* configuration, obtained by means of the algorithm configuration system *smac* (2.02.00; [13]), and
3. a multi-threaded *clasp* configuration using a portfolio of four competitively searching threads (cf. [14]); originally obtained by manual tuning and extensive experimentation; now available via the option `--configuration=chatty`.

Following common practice in automatic algorithm configuration, we selected the best outcome from ten independent runs of *smac* on a training set of instances relying on a different *Ricochet Robots* board than our benchmarks. Each *smac* run was allotted 100 hours for tuning 94 (discrete and continuous) parameters of *clasp*, using a cutoff of 600 seconds wall-clock time for *clasp*.

Table 1 provides average runtimes in seconds (accounting for timeouts by 600 seconds), absolute numbers of timeouts, and average times in seconds while penalizing timeouts by 6000 seconds (PAR10) over our 256 benchmark instances. We applied the aforementioned *clasp* configurations to solve the decision problem of plan existence as well as the optimization problem of shortest plan computation, relying on the plain encoding in Listing 2 or the advanced encoding in Listing 4, respectively.

Interestingly, the reference configurations with `berkmin` or `vsids`, respectively, perform significantly better with the plain than the advanced encoding. Analyzing the search statistics reported by *clasp*, on the one hand, we observed that the five times smaller ground instantiation size with the advanced encoding brings about the same amount of higher raw speed but, on the other hand, leads to roughly one order of magnitude more conflicts upon search. The trade-off between compactness and search efforts shifts towards the advanced encoding for the manually or automatically configured

	clasp w `berkmin`			*clasp* w `vsids`		
	Runtime	Timeout	PAR10	Runtime	Timeout	PAR10
clasp (decision)	227	71	1725	86	27	655
clasp (optimization)	326	114	2731	224	77	1848
unclasp	574	245	5742	567	242	5671
iclingo	229	83	1980	186	66	1578
oclingo	216	80	1903	179	62	1487

Table 2. Solving *Ricochet Robots* with different ASP systems

clasp settings, although the plain encoding generally still yields fewer conflicts.[7] Also note that automatic configuration via *smac* was accomplished relative to the advanced encoding. Thus, it performs worse than the manually selected configuration with the plain encoding. With the advanced encoding for which it has been tuned, *smac*'s configuration turns out as the best in single-threaded settings. In fact, it even surpasses multi-threaded settings regarding the number of timeouts in optimization, while the parallelism brought by multi-threading exhibits significantly improved robustness otherwise. The success of *smac* (with the advanced encoding) confirms analogous results in ASP [15, 16] and related areas [17, 18], showing that the burden of solver configuration can and should be taken off the user. Comparing decision and optimization problems, Table 1 further yields consistent relative performances of configurations, suggesting that underlying problem characteristics are quite similar in solving either kind of task.

3.2 ASP solving technologies

This section is dedicated to the empirical comparison of different ASP solving technologies. To this end, we concentrate on the advanced encoding given in Section 2.3. As points of reference, we consider *clasp* (1.3.10) for solving decision and optimization problems with a fixed `horizon` of 20 steps.[8] Running this version (rather than 2.1.3) is motivated by its usage in the ASP systems we compare: the *clasp* derivative *unclasp* (0.1; [19]), pursuing an unsatisfiability-based approach to optimization; the incremental ASP system *iclingo* (3.0.5; [10, 11]), performing iterative deepening by means of stepwise grounding and solving; and the reactive ASP system *oclingo* (3.0.92; [12]), extending *iclingo* with online capacities to solve sequences of queries. We benchmark all ASP systems in two settings, performing search with *clasp* in its default configuration (including `--heuristic=berkmin`) and the variant with `--heuristic=vsids`.

Table 2 shows experimental results, as before providing average runtimes, absolute numbers of timeouts, and average times penalizing timeouts by 6000 seconds (PAR10). Although *unclasp*'s approach to optimization can be highly effective (cf. [19]), it does

[7] Ground instantiations induce about 800k constraints with the plain encoding and 140k constraints with the advanced encoding. The average number of conflicts reported by *clasp* in its default configuration is 65k with the plain encoding and 510k with the advanced encoding. The latter number reduces to 127k conflicts on average in *smac*'s configuration, while no comparably substantial reductions are achieved with the plain encoding in any configuration.

[8] Grounding times of *gringo* are negligible, i.e., less than 0.2 seconds for our 16×16 board.

not work well for *Ricochet Robots*. In fact, *unclasp* aims at localizing substructures of a problem responsible for penalties within a **#minimize** statement. For the one in Listing 3, this means that atoms with predicate goon/1, indicating that some goal condition is not established at a time point, are gradually admitted to hold. Given that the establishment of goals, as for instance expressed by `target(red,5,2)`, relies on the whole trajectory from the initial time point `0`, reasons for penalties can hardly be subdivided into (independent) local substructures. The inherent causal connection between states in a planning problem thus undermines *unclasp*'s approach to optimization.

The incremental ASP system *iclingo* computes shortest plans in an unbounded fashion by gradually extending the horizon. To be more precise, starting at 1, *iclingo* grounds and solves (the incremental variant of) Listing 4 step by step until the first stable model, corresponding to a shortest plan, is obtained. Accordingly, the performance of *clasp* in optimization constitutes the reference for assessing *iclingo*, and the reduction of timeouts (31 with `--heuristic=berkmin` and 11 with `--heuristic=vsids`) shows the success of *iclingo*'s incremental approach. For one, this relies on the fact that stepwise grounding avoids the instantiation of rules for "unnecessary" time points. For another, recorded conflict information may be passed along between successive solving steps given that the solving component of *iclingo* remains in place until a plan is found. That is, grounding as well as solving efforts spent on unsatisfiable (decision) problems with too small horizons still contribute to and potentially foster progress in the sequel.

Going further beyond *iclingo*, the reactive ASP system *oclingo* maintains its solving component for dealing with consecutive target positions. In this way, recorded conflict information can be shared among all benchmark instances, which enables *oclingo* to exploit similarities in solving a series of planning problems. We thus obtain a reduction of timeouts in comparison to *iclingo* (3 with `--heuristic=berkmin` and 4 with `--heuristic=vsids`). However, note that *oclingo* does not decrease the planning horizon when a new target is entered since instantiations of rules for time points remain in the system once they have been produced in view of a query. As a consequence, a plan is not guaranteed to be shortest when its target position can be reached without incrementing the step counter. We aim at overcoming this in the future by extending incremental and reactive ASP solving to optimization, via which shortest plans could be addressed without withdrawing any formerly produced ground rules.

An alternative experiment performed with *oclingo* simulates "playing in rounds" by taking robot positions after achieving a goal as initial positions for the next target. Using the same sequence of targets as above, *oclingo* with `--heuristic=vsids` completed 250 instances in 26 seconds average runtime, thus exhibiting significant improvements over the setting with fixed initial positions. This phenomenon is probably related to the lexicographical order of our sequence of targets, and we aim at further experiments with less regular target sequences in the near future.

4 Discussion

Alex Randolph's board game *Ricochet Robots* offers a rich and versatile benchmark for ASP. As it stands, it represents a simple multi-agent planning problem in which each agent, i.e., robot, has limited sensing capacities (that is, only bumps are detected). This

setting leaves room for numerous interesting extensions. For example, we may consider competing or collaborating robots, simultaneous moves, and conceive compelling multi-agent scenarios. Also, the addition of resources like fuel or keys are conceptually interesting extensions, not to mention the plenty variants of the board game available on the web. Moreover, the potential of *Ricochet Robots* is even beyond ASP given that it can be modeled in many other paradigms, like action and planning languages, constraint languages, or in terms of Satisfiability testing.

In this paper, we started by elaborating upon two alternative encodings, one following traditional approaches to ASP planning and another centered around states rather than actions. More disparate encodings will result from the ASP competition in 2013, where *Ricochet Robots* is included in the modeling track. In addition, we provided the graphical tool *robotviz* for visualizing boards as well as solutions to the *Ricochet Robots* problem. The goal of this is to ease acquaintance with the game and to increase its attractiveness, also in view of teaching ASP. The visualization tool *robotviz* along with encodings and instances of *Ricochet Robots* are available at [20].

We illustrated the versatility of the benchmark by conducting two transverse empirical case studies. The first one aimed at assessing the impact of modeling techniques on the performance of ASP systems. This was flanked by an investigation of algorithm configuration and multi-threading as means to speed-up search, which demonstrated the capabilities of automatic solver configuration and parallelism. The second part of our study contrasted distinct ASP solving technologies in a uniform setting. Here, incremental and reactive ASP solving showed to be effective for computing shortest plans. Given that the original *Ricochet Robots* game proceeds in rounds, continuing from the final configuration of the previous round with a new target, automated support of such application scenarios in the future promises a rich source of reactive ASP benchmarks.

Our case studies are of course not sufficient for general claims but show the prospect of having a benchmark for evaluating different aspects in a uniform setting. This is certainly important in view of establishing a production mode for ASP when faced with singular real-world applications. In fact, the development of a robust ASP-based solution to an application problem must account for several interdependent factors and eventually converge to an integrated approach dealing with them. For one, the problem encoding predetermines the prospects of solving methods on problem instances, and its conception thus deserves careful consideration. Second, the application task designates appropriate solving methods, where decision and optimization as well as bounded and unbounded approaches can be distinguished. Finally, algorithm configuration and parallelism are powerful means to improve the efficiency of a solving method.

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

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)

3. 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)
4. Engels, B., Kamphans, T.: On the complexity of Randolph's robot game. Research Report 005, Institut für Informatik, Universität Bonn (2005)
5. Engels, B., Kamphans, T.: Randolph's robot game is NP-hard! Electronic Notes in Discrete Mathematics **25** (2006) 49–53
6. 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
7. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138**(1-2) (2002) 39–54
8. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. [21] 345–351
9. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. `http://potassco.sourceforge.net`
10. 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
11. Gebser, M., Kaminski, R., Schaub, T.: Gearing up for effective ASP planning. In Erdem, E., Lee, J., Lierler, Y., Pearce, D., eds.: Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz. Volume 7265 of Lecture Notes in Computer Science. Springer-Verlag (2012) 296–310
12. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. [21] 54–66
13. Hutter, F., Hoos, H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION'11). Volume 6683 of Lecture Notes in Computer Science. Springer-Verlag (2011) 507–523
14. Gebser, M., Kaufmann, B., Schaub, T.: Multi-threaded ASP solving with clasp. Theory and Practice of Logic Programming **12**(4-5) (2012) 525–545
15. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M., Ziller, S.: A portfolio solver for answer set programming: Preliminary report. [21] 352–357
16. Silverthorn, B., Lierler, Y., Schneider, M.: Surviving solver sensitivity: An ASP practitioner's guide. [22] 164–175
17. Hutter, F., Babić, D., Hoos, H., Hu, A.: Boosting verification by automatic tuning of decision procedures. In: Proceedings of the Seventh Conference on Formal Methods in Computer-Aided Design (FMCAD'07). IEEE Computer Society Press (2007) 27–34
18. Vallati, M., Fawcett, C., Gerevini, A., Hoos, H., Saetti, A.: Generating fast domain-specic planners by automatically configuring a generic parameterised planner. In: Proceedings of the Twenty-First ICAPS Workshop on Planning and Learning (PAL2011). (2011) 21–27
19. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. [22] 212–221
20. Potassco Labs: `http://potassco.sourceforge.net/labs.html`
21. 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)
22. Dovier, A., Santos Costa, V., eds.: Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12). Volume 17 of Leibniz International Proceedings in Informatics. Schloss Dagstuhl (2012)