Coala: A compiler from action languages to ASP

Coala: A Compiler from Action Languages to ASP

Martin Gebser, Torsten Grote, and Torsten Schaub*

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam

Abstract. Action languages allow for compactly describing dynamic domains. They are usually implemented by compilation, e.g., to Answer Set Programming. To this end, we developed a tool, called *Coala*, offering manifold compilation techniques for several action languages. We provide an overview of the salient and distinctive features of *Coala* as well as an experimental comparison of them.

1 Introduction

Action languages provide a compact formal model for describing dynamic domains [1], being central to many applications like model checking, planning, robotics, etc. Moreover, action languages can be implemented rather efficiently through compilation to Answer Set Programming (ASP; [2]) or Satisfiability Checking (SAT; [3]). Our system *Coala* takes advantage of this by offering a variety of different compilation techniques for several action languages.

Coala originates from *al2asp*, constituting the heart of the *BioC* system [4] used for reasoning about biological models in action language $\mathcal{C}_{TAID}$ [5]: *al2asp* compiles $\mathcal{C}_{TAID}$ to $\mathcal{C}$, which is in turn mapped to ASP via the transformation in [6]. *Coala* extends the capacities of *al2asp* in several ways. First, it adds certain features of $\mathcal{C}$+ [7] and provides full support of $\mathcal{B}$ [8] (and $\mathcal{A}_L$). Second, it offers different compilation schemes. Apart from a priori bounded encodings using standard ASP systems, *Coala* furnishes incremental encodings that can be used in conjunction with the incremental ASP system *iClingo* [9]. Moreover, *Coala* distinguishes among forward and backward (incremental) encodings, depending on whether trajectories are successively extended from initial states or whether they are built backwards starting from final states. Third, *Coala* supports all action query languages, $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$, in [1]. Fourth, *Coala* allows for posing LTL-like queries, following [10]. Finally, *Coala* offers the usage of first-order variables that are treated by the underlying ASP grounder. Optionally, type checking for variables can be enabled. *Coala* is implemented in `C++` and can also be used as a library; it is freely available at [11].

Our general approach is similar to the one taken by $DLV^{\mathcal{K}}$ [12] for processing action language $\mathcal{K}$. Similarly, $CCalc$ [7] addresses $\mathcal{C}$+ in its generality and maps it to SAT. We provide in [4] an empirical evaluation comparing *al2asp* (aka *Coala*'s bounded encoding of $\mathcal{C}$ following [6]) in combination with *Gringo* and *clasp* to $CCalc$ and $DLV^{\mathcal{K}}$ on benchmarks expressed in $\mathcal{C}$. Among the three systems, this rudimentary version of *Coala* had an edge in terms of robustness, having by far the fewest number of timeouts. Hence, in what follows we concentrate on the novel features of *Coala* going well beyond existing systems.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

2 Coala at Work

Coala starts with parsing an action description by means of an easily adaptable `bison++`-based parser before compiling it into a non-ground logic program. This program is grounded by *Gringo* and optionally extended by further ground facts before trajectories are generated via *clasp*. In what follows, we sketch the major compilation features of *Coala* illustrated by constructs of $\mathcal{C}$ [1].

To begin with, *Coala* generates via option `-e` either instance-based or direct encodings. For instance, the dynamic causal law

```
<caused> -alive <if> hit <after> shoot.
```

can be mapped onto either the fact `caused(neg(alive),hit,true,shoot).` or the rule `-fluent_alive(T) :- not -fluent_hit(T), action_shoot(T-1).` While a direct encoding is executable without further additions (cf. [6]), an instance-based encoding relies on meta-interpretation through an accompanying encoding. Although we do not detail it here, such meta-interpretation is very flexible and thus an easy way to implement different strategies.

Another major feature of *Coala* is the usage of incremental ASP solving techniques (via option `-i`), as provided by *iClingo*.[1] In this case, an action description is mapped onto a parametrized threefold logic program $B \cup P[k] \cup Q[k]$, among which $P[k]$ and $Q[k]$ contain a parameter k ranging over positive integers. Program B describes static knowledge, independent of k. The role of $P[k]$ is to capture knowledge accumulating with increasing k, whereas $Q[k]$ is specific for each value of k. The goal is then to compute answer sets of program $B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i]$ for some (minimum) integer $i \geq 1$. In an incremental setting, the above dynamic law is mapped onto

```
#cumulative t.
-fluent_alive(t) :- not -fluent_hit(t),action_shoot(t-1).
```

indicating that the rule belongs to $P[\mathtt{t}]$; its ground instances are successively produced and accumulated in the solver. Similarly, declarations `#base.` and `#volatile t.` indicate whether a rule belongs to B or $Q[k]$, respectively. Unlike this, a non-incremental setting is guarded by a fixed number of time steps `t`, provoking repetitive grounding of rules during iterative deepening search.

A third major feature is *Coala*'s distinction between forward and backward (incremental) encodings (via option `-r`), depending on whether trajectories are successively extended from initial states or whether they are built backwards starting from final states. This is implemented by means of meta-interpretation. To get an impression, consider the following three "meta-rules":

```
1 { holds(F,-t), holds(neg(F),-t) } 1 :- fluent(F).
fire(F,G,P,A,-t) :- caused(F,G,P,A), occurs(A,-t),
                    holds(P,-t), holds(G,-t+1).
:- fire(F,G,P,A,-t), not holds(F,-t+1).
```

The first rule aims at guessing a predecessor state (time stamp `-t`). The second one determines firing dynamic laws. Third, the integrity constraint ensures that the effects of firing causal laws are consistent with the successor state (time stamp `-t+1`).

[1] Note that *iClingo* relies on *Gringo* and *clasp*.

Benchmark	#	default time	default stime	-n time	-n stime	-i time	-i stime	-e time	-e stime	-e -i time	-e -i stime	-e -i -r time	-e -i -r stime
blocksworld/b08	16	44.19	20.90	44.31	21.84	4.30	4.23	323.43	69.98	76.83	74.83	112.95	109.78
blocksworld/b09	16	53.56	9.27	55.78	9.58	11.68	11.58	TO	TO	283.90	280.33	338.64	332.95
blocksworld/b10	17	88.69	20.53	77.38	11.74	14.69	14.56	TO	TO	366.34	361.42	MO	MO
blocksworld/b11	18	403.31	276.31	403.78	247.84	117.53	117.37	TO	TO	TO	TO	MO	MO
blocksworld/b12	19	228.47	160.37	290.91	163.54	69.99	69.81	MO	MO	MO	MO	MO	MO
ferryman/f03	15	29.38	6.80	29.79	7.26	2.67	2.63	49.84	7.43	7.40	6.99	13.09	12.31
ferryman/f04	17	65.40	14.48	64.00	13.26	3.48	3.44	96.89	14.20	11.84	11.17	26.96	25.71
ferryman/f05	19	132.22	26.67	100.30	26.83	4.36	4.32	170.14	23.46	19.35	18.31	46.27	44.35
ferryman/f06	17	122.92	28.63	120.93	26.29	6.03	5.97	202.99	28.12	26.57	25.26	67.57	65.13
ferryman/f07	19	243.05	54.12	257.18	50.53	18.04	17.96	356.48	48.50	40.12	38.19	138.62	135.09
hanoi/h03	31	85.89	9.77	88.48	10.68	1.83	1.80	50.80	5.01	4.66	4.50	281.18	280.86
hanoi/h04	63	TO	TO	TO	TO	69.51	69.41	TO	TO	54.74	54.19	TO	TO
yale/y04	18	6.22	0.16	6.14	0.20	10.18	10.14	11.35	0.07	5.13	5.10	16.11	16.08
yale/y05	20	40.91	0.14	35.52	0.79	16.83	16.80	64.19	0.86	40.59	40.56	69.95	69.91
yale/y06	22	132.51	5.88	165.62	4.50	79.05	79.02	87.20	0.34	177.14	177.10	TO	TO
yale/y07	24	547.40	0.45	TO	TO	205.35	205.31	499.40	22.02	TO	TO	TO	TO
Average (Outs)		176.51 (1)		183.76 (2)		39.72 (0)		307.04 (5)		182.16 (3)		294.46 (6)	

Table 1. Experiments comparing different target compilations of *Coala*.

Moreover, *Coala* supports LTL-like queries, using *next*, *finally*, *globally*, *until*, *weak until*, and *release*, viz. `X, F, G, U, W, R`, and aims at generating counterexamples. For instance, the simple LTL query `LTL: X alive.` asks whether the fluent `alive` is true in the next step in all trajectories. Following [10], this is translated to

```
ltl_counter_example :- ltl_f_2(0).
ltl_f_2(0) :- -fluent_alive(1).
```

producing counterexamples in which the complement of `alive` holds. More complex LTL formulas require additional auxiliary rules and are omitted here for brevity.

Finally, a typical call of *Coala* looks as follows:

```
coala -l b -i bw.alb | cat - bw.stat | iclingo 0
```

The options '`-l b -i`' tell *Coala* that `bw.alb` is written in $\mathcal{B}$ and that it should be compiled into an incremental ASP program. The latter is then augmented with static domain knowledge in `bw.stat` before *iClingo* is invoked to compute all answer sets for a minimum number of time steps. The interested reader is directed to [11] for more details on the language and usage of *Coala*.

3 Experiments

We conducted experiments in order to evaluate the different compilation techniques furnished by *Coala*. To this end, we confined ourselves to action language $\mathcal{C}$ and concentrate on combinations of several *Coala* options: '`-n`' enables dedicated handling of classical negation; '`-i`' produces an incremental encoding for *iClingo*; '`-e`' uses meta-interpretation (rather than direct encoding); '`-r`' uses backward encoding. The `default` setting includes none of these features. All experiments were run with *iClingo* (2.0.5), using *clasp* (1.3.2) in its default settings on an Intel Core 2 Duo CPU at 2.66 GHz running Ubuntu GNU/Linux 9.10 with RAM usage limited to 1.5 GB. All programs were run sequentially as single threads on one CPU core.

Our results are summarized in Table 1. The first two columns give the respective benchmark along with its horizon (#). Note that the next three columns use direct en-

codings, while the last three rely upon meta-interpretation. Column *time* is average CPU time from three runs per benchmark; *stime* is average time needed by the solver (in the final successful run during iterative deepening search; and in total for `-i`). An entry TO indicates timeout after 600 seconds, while MO means that the processes were aborted at 1.5 GB RAM consumption. The last row shows the average CPU time (and number of timeouts) over all benchmarks. In case of timeout, a time of 600 seconds was assumed.

Looking at the global outcome in the last row, we observe that incremental direct encodings (`-i`) perform best over all benchmarks (except for *h04* and *y04*). Although worse, the incremental non-direct counterpart (`-e -i`) performs best on average among the meta-interpreted encodings. Changing to a more complex backward encoding (`-e -i -r`) does not lead to an improvement and yields two more memory exhaustions than the other meta-interpreted encodings. Pure meta-interpretation (`-e`), suffering from a grounding overhead, performs worst, despite of solving one more instance than the backward encoding. No clear difference was observable on the usage of built-in classical negation (`-n`), producing more integrity constraints than a dedicated treatment (`default`).

All in all, we observe that an incremental approach to action languages is largely beneficial. The usage of backward encodings may make a difference on particular problem classes. Although meta-interpretation appears to lead to less efficient encodings, it offers an easy way to experiment with different strategies.

Acknowledgments. This work was partially funded by BMBF project GoFORSYS and DFG grant SCHA 550/8-1.

References

1. Gelfond, M., Lifschitz, V.: Action languages. Electronic Transactions on Artificial Intelligence **3**(6) (1998) 193–210
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University (2003)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. IOS (2009)
4. Dworschak, S., Grote, T., König, A., Schaub, T., Veber, P.: The system BioC for reasoning about biological models in $\mathcal{C}$. In: ICTAI'08. IEEE (2008) 11–18
5. Dworschak, S., Grell, S., Nikiforova, V., Schaub, T., Selbig, J.: Modeling biological networks by action languages via ASP. Constraints **13**(1-2) (2008) 21–65
6. Lifschitz, V., Turner, H.: Representing transition systems by logic programs. In: LPNMR'99. Springer (1999) 92–106
7. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153**(1-2) (2004) 49–104
8. Son, T., Baral, C., Nam, T., McIlraith, S.: Domain-dependent knowledge in answer set planning. ACM Transactions on Computational Logic **7**(4) (2006) 613–657
9. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: ICLP'08. Springer (2008) 190–205
10. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming **3**(4-5) (2003) 519–550
11. `http://potassco.sourceforge.net`
12. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning. Artificial Intelligence **144**(1-2) (2003) 157–211