Efficiently Implementing GOLOG with Answer Set Programming

Malcolm Ryan
School of Computer Science and Engineering
University of New South Wales, Australia
malcolmr@cse.unsw.edu.au

Abstract

In this paper we investigate three different approaches to encoding domain-dependent control knowledge for Answer-Set Planning. Starting with a standard implementation of the action description language $\mathcal{B}$, we add control knowledge expressed in the GOLOG logic programming language. A naive encoding, following the original definitions of Levesque et al., is shown to scale poorly. We examine two alternative codings based on the transition semantics of ConGOLOG. We show that a speed increase of multiple orders of magnitude can be obtain by compiling the GOLOG program into a finite-state machine representation.

Action programming languages have a long history of use as a form of domain-dependent control knowledge in planning (Thielscher 2008). They provide a means to prune the search space of a planner by allowing the user to specify high-level plan structure while leaving space for the planner to fill in the specific details. Many different languages have been proposed for this purpose, including temporal logic (Doherty and Kvarnstom 1999; Bacchus and Kabanza. 2000) and hierarchical task networks (Nau et al. 1999). One of the more enduring languages is GOLOG. Originally designed by Levesque et al. (1997) for use with the situation calculus (Reiter 2001), it has since been employed with other planning systems, including FF (Claßen et al. 2007) and Answer Set Planning (Son et al. 2006).

In this paper we are interested in this latter problem: implementing GOLOG as an answer set program (ASP) with the action description language $\mathcal{B}$ (Gelfond and Lifschitz 1998). While Son et al. have already addressed this problem, we found that their implementation scaled poorly to large programs. This is a common issue with answer set programming: while the declarative semantics make specifying a problem simple, naive problem descriptions often give rise to inefficient solutions. Often a different encoding can change the solution time by orders of magnitude (Gebser et al. 2013). Particular attention has to be paid to the grounding process, whereby first order logic programs are turned into propositional logic. We find that the direct encoding of the GOLOG definition often produces grounded ASPs that increase polynomially in size with the length of the GOLOG program. A different encoding, based on the finite-state machine semantics of ConGOLOG (Giacomo, Lespérance, and Levesque 2000) can in many cases reduce this to linear growth.

In what follows we first describe the basics of GOLOG and Son et al.'s original ASP encoding, then two alternatives based on ConGOLOG and finite state machines. We calculate the size of the grounded ASP for basic program structures and then we empirically evaluate each approach on a complex example program. We conclude that our two new encodings are generally superior to the original, although the best implementation is still problem dependent.

GOLOG and ConGOLOG

GOLOG was originally expressed in terms of the situation calculus, but for greater generality we shall consider it simply in terms of a set of constraints on a plan expressed as an interleaved sequence of states and actions. We define the predicates $holds(T, \phi)$ to denote that some formula ϕ holds at time T and $does(T, a)$ to denote that the agent does action a at time T. We assume that the standard planning constraints between states and actions hold. The details of these constraints do not effect the definitions below.

A simple GOLOG program consists of six basic elements:

the empty program: nil

primitive actions: $a, b \in A$

tests: ϕ? for a formula ϕ describing a set of states

sequences: $\delta_1; \delta_2$ for programs δ_1 and δ_2

choices: $\delta_1 | \delta_2$ for programs δ_1 and δ_2

repeat: δ^* for program δ

The complete language description also includes choice over variable instantiation and procedure calls. We have omitted them here for reasons of space. They do not significantly impact the results that follow.

The semantics of these operators are described by the predicate $do(T_1, T_2, P)$ which designates that program P is started at time T_1 and is completed at time T_2. We can encode these semantics in ASP as shown in Figure 1.[1].

[1]In their paper, Son et al. call this predicate *trans* rather than *do*

$$
\begin{aligned}
do(T,T,nil) &\leftarrow time(T).\\
do(T,T+1,action(A)) &\leftarrow time(T),\ program(action(A)),\ does(T,A).\\
do(T,T,test(F)) &\leftarrow time(T),\ program(test(F)),\ holds(T,F).\\
do(T_1,T_2,seq(A,B)) &\leftarrow time(T_1),\ time(T_2),\ program(seq(A,B)),\ T_1 \le T \le T_2,\ do(T_1,T,A), do(T,T_2,B).\\
do(T_1,T_2,choose(A,B)) &\leftarrow time(T_1),\ time(T_2),\ program(choose(A,B)),\ do(T_1,T_2,A).\\
do(T_1,T_2,choose(A,B)) &\leftarrow time(T_1),\ time(T_2),\ program(choose(A,B)),\ do(T_1,T_2,B).\\
do(T,T,repeat(A)) &\leftarrow time(T),\ program(repeat(A)).\\
do(T_1,T_2,repeat(A)) &\leftarrow time(T_1),\ time(T_2),\ program(repeat(A)),\\
&\qquad T_1 < T_m \le T_2,\ do(T_1,T_m,A),\ do(T_m,T_2,repeat(A)).
\end{aligned}
$$

Figure 1: An encoding of GOLOG's *do* predicate in ASP.

An important factor in this encoding is that ASP solvers generally only work over finite domains. This means that each variable needs to be 'safe', i.e. it must be grounded over a finite domain. The recursive definition of the GOLOG language above allows an unbounded grounding for the program argument to the *do* predicate. To prevent this from causing a problem, we introduce the *program* predicate, which, given a starting program, enumerates all subparts of the program (Figure 2). This provides a finite domain for the program variable while also allowing all possible subprograms that might be encountered during execution.

One of the guidelines for writing more efficient ASP programs is to keep the number of ground rules and atoms low (Gebser et al. 2013). The *do* predicate in this encoding is defined in terms of two time points, T_1 and T_2. We should expect therefore (as we verify later) that grounding program will produce a number of rules quadratic in the number of time points as ground rules can possibly be generated for every pair (T_1, T_2). This is liable to increase both grounding and solving time for these problems.

We would prefer an encoding which only takes a single time argument. To this end, we look to ConGOLOG (Giacomo, Lespérance, and Levesque 2000), an alternative encoding of the language designed to include concurrency. We won't be addressing the concurrent aspects in this paper, but we are interested in the alternative execution semantics. In place of the *do* predicate ConGOLOG defines predicates $trans(T, P_1, P_2)$ and $final(T, P)$. The *trans* predicate states that program P_1 can be executed for a single time step at time T, leaving program P_2 remaining to be executed. P_2 is termed the 'continuation' of P_1. The *final* predicate states that program P can be considered finished at time T without further execution. We can encode this in ASP as shown in Figure 3.[2]

but we prefer to keep the original naming to avoid confusion with the definition of *trans* from ConGOLOG.

$$
\begin{aligned}
program(A) &\leftarrow program(seq(A,B)).\\
program(B) &\leftarrow program(seq(A,B)).\\
program(A) &\leftarrow program(choose(A,B)).\\
program(B) &\leftarrow program(choose(A,B)).\\
program(A) &\leftarrow program(repeat(A)).
\end{aligned}
$$

Figure 2: Generating all subprograms from a given program.

Once again, we need some means to ground the program variables in this encoding. The earlier definition of the *program* predicate is insufficient, as the *trans* predicate creates new program fragments that do not appear as subprograms of the original. Consider, for example, the program:

$$repeat(seq(a,b))$$

If we execute this for one time step by performing the action a the continuation of this program is:

$$seq(b, repeat(seq(a,b)))$$

We need to include this continuation as a possible grounding of any program variable. This can be achieved with the rule:

$$program(P') \leftarrow program(P), continue(P,P').$$

and a definition for *continue* for similar to that of *trans* in Figure 3 with the timestamp argument and references to *holds* and *does* removed (we omit the details for the sake of brevity).

With this definition of *trans* we can redefine *do* as:

$$
\begin{aligned}
do(T,T,P) &\leftarrow T = n, final(T,P).\\
do(T_1,T_2,P) &\leftarrow time(T),\ T_2 = n,\ program(P),\\
&\qquad trans(T_1,P,P'),\ do(T_1+1,T_2,P').
\end{aligned}
$$

[2] Note there is one slight variation between this and the standard ConGOLOG definition. In ConGOLOG the handling of the test command $\phi?$ needs special attention because it is possible to interleave tests and actions from different concurrent threads. As we are not implementing concurrency in this paper, we can simplify the definition.

$$\begin{aligned}
\mathit{trans}(T, \mathit{action}(A), \mathit{nil}) &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{action}(A)),\ \mathit{does}(T, A).\\
\mathit{trans}(T, \mathit{seq}(A, B), \mathit{seq}(A', B)) &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{seq}(A, B)),\ \mathit{trans}(T, A, A').\\
\mathit{trans}(T, \mathit{seq}(A, B), B') &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{seq}(A, B)),\ \mathit{final}(T, A),\ \mathit{trans}(T, B, B').\\
\mathit{trans}(T, \mathit{choose}(A, B), A') &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{choose}(A, B)),\ \mathit{trans}(T, A, A').\\
\mathit{trans}(T, \mathit{choose}(A, B), B') &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{choose}(A, B)),\ \mathit{trans}(T, B, B').\\
\mathit{trans}(T, \mathit{repeat}(A), \mathit{seq}(A', \mathit{repeat}(A))) &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{repeat}(A)),\ \mathit{trans}(T, A, A').\\
\\
\mathit{final}(T, \mathit{nil}) &\leftarrow \mathit{time}(T).\\
\mathit{final}(T, \mathit{test}(F)) &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{test}(F)),\ \mathit{holds}(T, F).\\
\mathit{final}(T, \mathit{seq}(A, B)) &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{seq}(A, B)),\ \mathit{final}(T, A),\ \mathit{final}(T, B).\\
\mathit{final}(T, \mathit{choose}(A, B)) &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{choose}(A, B)),\ \mathit{final}(T, A).\\
\mathit{final}(T, \mathit{choose}(A, B)) &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{choose}(A, B)),\ \mathit{final}(T, B).\\
\mathit{final}(T, \mathit{repeat}(A)) &\leftarrow \mathit{time}(T),\ \mathit{program}(\mathit{repeat}(A)).
\end{aligned}$$

Figure 3: An encoding of ConGOLOG's $trans$ predicate in ASP.

While we have maintained the arity three predicate for clarity, the second time argument is always bound to n, the maximum time step of our plan, so the grounding of this predicate is now linear in the plan duration.

We need also consider the grounding of the *trans* and *final* predicates. The *trans* could potentially blow-up if a subprogram has a lot of possible continuations, but this seems unlikely in practice. Relatively few programs are can be completed without any action so the *final* predicate is also unlikely to cause problems.

There is, however, another issue: the ground ASP will contain a *trans* fact for almost every subprogram at every time point, which can again cause quadratic growth in the grounding. We can avoid this by eliminating any explicit reference to sub-programs altogether.

Compilation to a Finite State Machine

It is no surprise the GOLOG programs resemble regular expressions. GOLOG is effectively a grammar over action sequences (with the addition of test fluents to enforce state properties). The grammar is regular[3] and so can be compiled into a finite state machine (FSM) (Thompson 1968). The states of the machine are the program and its continuations. Each edge is labelled with an action to be performed and a set of fluents that must be true to make the transition. The ASP encoding of Figure 3 implicitly represents this machine, but we can encode it explicitly as *state* and *goal* predicates.

The full algorithm is too long to include here, but we illustrate the concept with an example. Consider the program:

$$(\phi? \ ; \ a \ ; \ (b \mid c))^* \ ; \ \neg\phi? \ ; \ d \ ; \psi?$$

[3]Unless we allow recursive procedure calls, in which case it is context-free.

Let this be s_0, the initial state of our machine. There are two possible transitions from here. If ϕ is true, the agent may execute action a, with continuation:

$$(b \mid c) \ ; \ (\phi? \ ; \ a \ ; \ (b \mid c))^* \ ; \ \neg\phi? \ ; \ d \ ; \psi?$$

Let this be state s_1.

Otherwise, if $\neg\phi$ is true, the agent may execute action d and be left with the program:

$$\psi?$$

Let this be state s_2.

There are two transitions from state s_1, executing either b or c, both of which lead back to s_0. Finally s_2 is the only finishing state, under condition that ψ is true. This yields the augmented state machine shown in Figure 4(a).[4]

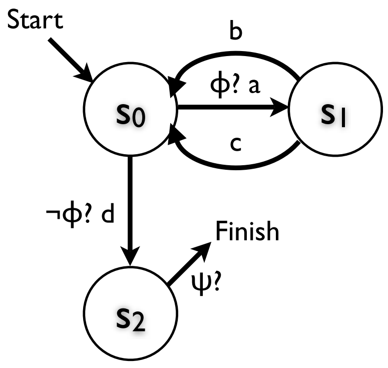
Having performed this compilation, we can encode the resulting state machine in ASP with individual rules describing each transition, as in Figure 4(b). The advantage of this representation is that we have eliminated all reference to subprograms, so we should expect a much smaller ground ASP. Furthermore we have eliminated the need to refer to programs as deeply nested functors, replacing them with numbered states. This fact alone provides a major boost in efficiency.

Grounded program sizes

We shall look at the size of the grounded ASP for each of the above approaches on the following GOLOG programs:

- The sequence: $a_1 \ ; \ a_2 \ ; \ \dots \ ; \ a_n$
- The choice: $a_1 \mid a_2 \mid \dots \mid a_k$

[4]Note that the machine is potentially non-deterministic if a program such as $(a \ ; \ b) \mid (a \ ; \ c)$ includes multiple transitions with the same state and action requirements but different continuations.



(a)

$$
\begin{aligned}
&state(0,0).\\
&state(T+1,1) \leftarrow time(T),\ state(T,0),\\
&\qquad\qquad holds(T,\phi),\ does(T,a).\\
&state(T+1,2) \leftarrow time(T),\ state(T,0),\\
&\qquad\qquad holds(T,\neg\phi),\ does(T,d).\\
&state(T+1,0) \leftarrow time(T),\ state(T,1),\ does(T,b).\\
&state(T+1,0) \leftarrow time(T),\ state(T,1),\ does(T,c).\\
&goal(T) \leftarrow time(T),\ state(T,2),\ holds(T,\psi).
\end{aligned}
$$

(b)

Figure 4: (a) The augmented state machine for the GOLOG program $(\phi? \ ; \ a \ ; \ (b \mid c))^* \ ; \ \neg\phi? \ ; \ d \ ; \psi?$ and (b) its ASP encoding.

- The nested repetition: $(\ldots((a^*)^*)^* \ldots)^*$

We used the *gringo* grounder from the *Potassco* ASP toolkit (Gebser et al. 2011) to ground the programs. The results are summarised in Table 1. We consider each in detail below,

Sequence: $a_1 \ ; \ a_2 \ ; \ \ldots \ ; \ a_n$

GOLOG: Since we encode sequences using recursive binary functors there are $O(n)$ subprograms for a sequence of length n. For each subsequence of length l there will be $n-l+1$ ground instances of the $do/3$ rule, giving $O(n^2)$ in total.
ConGOLOG: If we define the sequence operator as right-associative (i.e. $a \ ; \ b \ ; \ c \equiv a \ ; \ (b \ ; \ c)$) then the subprograms and the continuations of the sequence are identical. Otherwise there may be up to twice as many $program/1$ atoms as in GOLOG. In either case, the number is still $O(n)$.

There also are $O(n)$ ground instances of the $do/3$ rule, one at each time point, but there are now $O(n^2)$ instances of $trans/3$ as there is an instance of $trans$ for every subsequence at every time point.
FSM: There are $O(n)$ states in the state machine and each is connected by a single edge to the next. Notably, the grounder is able to recognise that there is a single reachable state for each time point, so there are only $O(n)$ ground instances of $state/3$.

Choice: $a_1 \mid a_2 \mid \ \ldots \ \mid a_k$

GOLOG: As with the sequence above, a choice of k arguments yields $O(k)$ $program/3$ facts due to the use of a binary *choose* functor. As the program executes in a single time step there are only $O(k)$ instances of the $do/3$ predicate, one for each action.
ConGOLOG: Continuations do not add any extra programs in this case, so the ground instances of $program/3$ are the same as for GOLOG. There is only one ground instance of the $do/3$ rule and $O(k)$ instances of $trans$ one for each subprogram.
FSM: There are two states joined by k edges. This results in $O(k)$ ground instances of $state/3$.

Nested repetition: $(\ldots((a^*)^*)^* \ldots)^*$

GOLOG: There are $O(d)$ instances of $program/3$ one for each level of nesting. The size of the $do/3$ rule grows quite quickly. Every subprogram is possible between every pair of time points. What is more, the midpoint time T_m can be ground to any value between T_1 and T_2 resulting in $O(n^3 d)$ grounded rules.
ConGOLOG: Let us write $a^{*(i)}$ to indicate nested repeats of depth i. Each subprogram $a^{*(i)}$ has continuation:

$$a^* \ ; \ a^{**} \ ; \ \cdots \ ; \ a^{*(i)}$$

Each of these programs is its own continuation, so there are $O(d)$ ground instances of the $program/3$ predicate. For each program of the form $a^{*(i)}$ there is a single ground $trans$ rule at each time step; for each program of the form $a^* \ ; \ a^{**} \ ; \ \cdots \ ; \ a^{*(i)}$ there are two (one for each case of the sequence rule). This yields a total of $O(nd)$ ground instances. Similarly there is an instance of the $do/3$ rule for each program at each time point, $O(nd)$ in all.
FSM: The nested repeats are lost in the compilation process, leaving two states joined by a single edge. This results in $O(n)$ ground instances of $state/3$.

From this analysis we would expect to some small improvements in performance in the ConGOLOG encoding over the GOLOG encoding and more significant improvements when the finite state machine implementation is used.

Testing

We implemented a compiler which output these three ASP encodings for a given GOLOG program, along with an implementation of the action description language $\mathcal{B}$. These programs were passed to a grounder and then an ASP solver. We use the *Potassco* tools *gringo* and *clasp* for this purpose[5]. To evaluate the encodings we ran different size instances of the program:

$$\big(a\big|((a|b);a) \ \big| \ ((a|b)^2;a) \ \big| \ \ldots \ \big| \ ((a|b)^{k-1};a)\big)^* \ ;(a|b)^k$$

[5]Experiments were run with *gringo* 3.0.5 and *clasp* 2.1.5. Available at http://potassco.sourceforge.net/

Program	Length	Actions	Approach	Atoms	Rules
$a_1 \,;\, a_2 \,;\, \ldots \,;\, a_n$	n	n	GOLOG	$program/1 : O(n)$	$do/3 : O(n^2)$
			ConGOLOG	$program/1 : O(n)$	$do/3 : O(n)$
					$trans/3 : O(n^2)$
					$final/2 : O(1)$
			FSM		$state/3 : O(n)$
$a_1 \mid a_2 \mid \ldots \mid a_k$	1	k	GOLOG	$program/1 : O(k)$	$do/3 : O(k)$
			ConGOLOG	$program/1 : O(k)$	$do/3 : O(1)$
					$trans/3 : O(k)$
					$final/2 : O(1)$
			FSM		$state/3 : O(k)$
$\underbrace{(\ldots((a^*)^*)^*\ldots)^*}_{\text{depth } d}$	n	1	GOLOG	$program/1 : O(d)$	$do/3 : O(n^3 d)$
			ConGOLOG	$program/1 : O(d)$	$do/3 : O(nd)$
					$trans/3 : O(nd)$
					$final/2 : O(nd)$
			FSM		$state/3 : O(n)$

Table 1: The size of the ground ASP for various programs and approaches.

where δ^k is a shorthand for a sequence of k instances of δ. This particular program was chosen as it shows exponential blow-up when converted to a deterministic finite automaton (Jirásek, Jirásková, and Szabari 2007). We wish to see how this growth affected each of our encodings. The action model contained only the actions a and b necessary for this program. Their preconditions were always true and they had no effects. This test is designed to focus on the variation in run-time caused by the GOLOG implementation, so the planning domain itself is deliberately simple.

We ran instances of this problem with size k varying from 5 to 100, measuring compilation, grounding and solving times[6]. In each case the maximum plan length was 100 time steps. Each instance was run 100 times and average times are reported. Runs that took more than 100s were aborted.

The total run time, including compilation, grounding and solving, for each approach is plotted in Figure 5(a). The difference in performance is striking. The ConGOLOG encoding is almost an order of magnitude faster than the GOLOG encoding and the finite state machine encoding is another order of magnitude faster still.

The size of the ground programs, as shown in Figure 5(b), also reflect this. The GOLOG program has a large number of ground rules and grows almost linearly. The ConGOLOG program has slower initial growth but increases quadratically in both rules and atoms. The finite state machine program is much smaller and grows much more slowly, at an apparently linear rate. It is interesting to note that while the ConGOLOG ground program ultimately grows larger than the GOLOG program, the total run time is still significantly shorter. The reason for this is not apparent.

It is informative to visualise the breakdown on this time into compilation, grounding and solving, as shown in Figure 6. For both GOLOG and ConGOLOG compilation is simply writing the program to a file and takes little time. For the finite state machine it is much more significant, taking an increasing proportion of the total time as the problem size increases. Grounding dominates for ConGOLOG and larger GOLOG problems. Our investigation seems to indicate that a lot of this time is due to the handling of deeply nested functors in program expressions. Possibly a more efficient method of handling such expressions could make these approaches more viable.

Space does not permit the presentation of further results but we have run similar tests with a variety of 'toy' problems of this nature and seen similar patterns of results – the finite state machine representation is always much faster than the other approaches. We hope to soon be able to test on more realistic problems but that will require the implementation of additional language features, concurrency being the most notable.

Conclusion

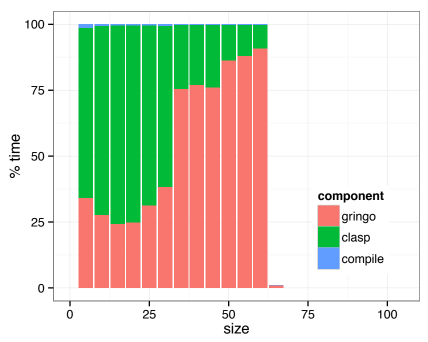
Efficient answer-set programming is still a difficult art. We have demonstrated three different encodings of the GOLOG action language, two based on the logical definitions of GOLOG and ConGOLOG and one based on a compiled finite-state machine representation. Our aim was to improve the run time of the resulting ASP by reducing the overall size of the ground program. Our results show that we succeeded in this aim. By compiling the GOLOG program to a finite state machine we have produced an encoding that runs approximately two orders of magnitude faster than the original encoding.

In future work we plan to expand these encodings to include the concurrent action semantics of the ConGOLOG language.

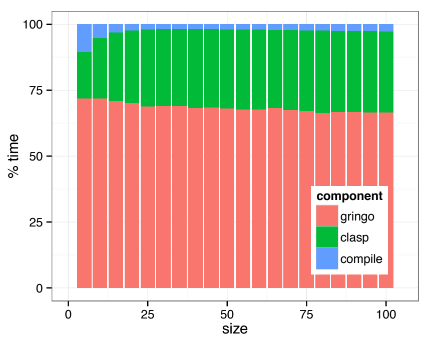
References

Bacchus, F., and Kabanza., F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1,2):123–191.

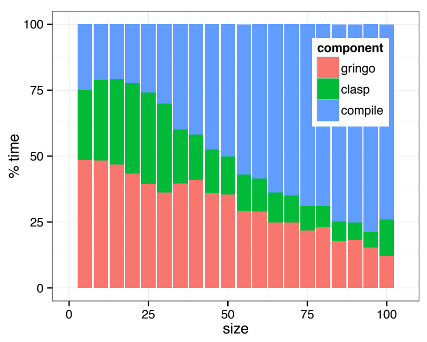
[6]Measurements were taken on a 3.20GHz Intel Xeon$^{\text{TM}}$.



(a) GOLOG

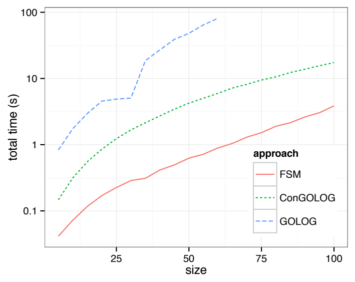


(b) ConGOLOG

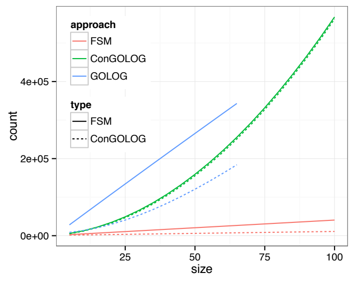


(c) FSM

Figure 6: Break down of run times for each approach.



(a) Total run time.



(b) The number of ground rules and atoms.

Figure 5: Total run time and size of ground programs for each approach. Time differences between runs were negligible ($< 2\%$), so error bars have been omitted.

Claßen, J.; Eyerich, P.; Lakemeyer, G.; and Nebel, B. 2007. Towards an Integration of Golog and Planning. In *Proceedings of the International Joint Conferences on Artificial Intelligence*.

Doherty, P., and Kvarnstom, J. 1999. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In *Proceedings of the 6th International Workshop on the Temporal Representation and Reasoning*.

Gebser, M.; Kaufmann, B.; Kaminski, R.; Ostrowski, M.; Schaub, T.; and Schneider, M. 2011. Potassco: The Potsdam Answer Set Solving Collection. *AI Communications* 24(2):107–124.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2013. *Answer Set Solving in Practice*. Morgan & Claypool Publishers.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on AI* 3(16).

Giacomo, G. D.; Lespérance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121:109–169.

Jirásek, J.; Jirásková, G.; and Szabari, A. 2007. Deterministic Blow-Ups of Minimal Nondeterministic Finite Automata over a Fixed Alphabet. *Developments in Language Theory, Lecture Notes in Computer Science Volume* 4588:254–265.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming* 31(1-3):59–83.

Nau, D.; Cao, Y.; Lotem, A.; and Munõz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the 16th International Conference on Artificial Intelligence.*, 968–973. Philadelphia, PA: AAAI Press.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Son, T. C.; Baral, C.; Tran, N.; and McIlraith, S. 2006. Domain-Dependent Knowledge in Answer Set Planning. *ACM Transactions on Computational Logic* 7(4):613–657.

Thielscher, M. 2008. *Action Programming Languages*. Morgan & Claypool Publishers.

Thompson, K. 1968. Regular Expression Search Algorithm. *Communications of the ACM* 11(6):419–422.