Online Agent Logic Programming with oClingo

Timothy Cerexhe[1], Martin Gebser[2,⋆], and Michael Thielscher[1]

[1] School of Computer Science and Engineering
The University of New South Wales, Sydney, Australia
[2] Helsinki Institute for Information Technology HIIT
Aalto University – Department of Information and Computer Science, Helsinki, Finland
{timothyc,mit}@cse.unsw.edu.au, martin.gebser@aalto.fi

Abstract. The online answer set solver oClingo offers a powerful new technique for uniting the speed of Answer Set Programming (ASP) with dynamic events. The price of this power is paid by increased constraints on the construction of a 'safe' program—one that satisfies an arcane modularity condition. We provide an alternative in the form of so-called Agent Logic Programs—a concise declarative language for describing agent control strategies. Specifically, we take an ASP-compatible subset of Agent Logic Programs, extend it with exogenous actions, argue this translation is faithful to the original definition, and prove that it guarantees oClingo's modularity condition. The result is a safe, clean input language for oClingo and a new implementation for Agent Logic Programs.

1 Introduction

The online answer set solver oClingo offers a powerful new technique for uniting the speed of answer set solving with dynamic events (Gebser et al., 2011). Unfortunately the design of this system required a restrictive modularity condition to guarantee that online answer sets could be efficiently computed. This modularity condition bears no resemblance to a program written in oClingo's input language, making it a challenging task for programmers to check that this crucial property actually holds. Individual successful runs offer no guarantees regarding this property either, and programs that ignore it risk incorrect results or an error message that provides limited assistance in identifying and correcting the violation. In response to this problem we propose the use of an alternate input language to oClingo that is safe (ie. guarantees modularity) but not at the expense of seriously impeding expressiveness. The online dimension of oClingo suggests an agent control language as a suitable substitute.

Action theories like the classical Situation Calculus (McCarthy and Hayes, 1969) provide the foundations for the design of artificial, intelligent agents capable of reasoning about their own actions. Several mature calculi now exist, including the Event Calculus (Mueller, 2006) and the modern Situation Calculus (Reiter, 2001), each capable of endowing agents with knowledge of complex dynamic environments. Despite these virtues, surprisingly few attempts have been made to integrate elaborate action theories into actual programming languages for intelligent agents. Most languages, such as

⋆ Also affiliated with the University of Potsdam, Germany, and supported by the Finnish Centre of Excellence in Computational Inference Research (COIN) under AoF grant #251170.

(Dastani et al., 2003; Morley and Myers, 2004; Bordini et al., 2007), prefer very limited reasoning with STRIPS-style updates (Fikes and Nilsson, 1971). Programmers of these languages must include belief updates as part of their behavioral strategies rather than keeping the program independent of the action theory.

Two notable exceptions to this trend are GOLOG (Levesque et al., 1997) and FLUX (Thielscher, 2005), procedural languages built on top of the Situation Calculus and Fluent Calculus, respectively. This construction, however, binds each language to its chosen action theory, and no program can be swapped with an alternate theory as the domain grows or requirements change. The procedural paradigms adopted by such programs can also cause friction against the declarative action theories, rendering declarative languages a valuable alternative (Lloyd, 1987).

Agent Logic Programs were proposed in response to these limitations (Drescher et al., 2009), offering a concise, declarative language for describing agent control strategies, independent of an action theory. In fact, such programs are action theory agnostic—any theory satisfying a few, general requirements is compatible with an Agent Logic Program. Moreover, this is achieved with just two additional operators over plain logic programming: "*do*" for executing actions, and "?" for checking conditions in the underlying theory. Like most predecessors though, Agent Logic Programs do not yet have an implementation based on the efficient Answer Set Programming technique. Indeed the only existing implementation, ALPprolog, uses prime implicants to solve the entailment problem (Drescher and Thielscher, 2011).

We resolve oClingo's deficiency with an automatic translation from Agent Logic Programs to oClingo, argue that this translation is faithful to the original definition and prove that it guarantees the modularity condition. Thus Agent Logic Programs serve as an intuitive interface to oClingo, effectively bypassing the demands placed on oClingo's users. In return, oClingo provides a new, online implementation.

Our results are threefold: First, we define the concept of *online* agent logic programs. Second, we present a translation of these OALPs to oClingo programs. Finally, we show that the result is guaranteed to satisfy the modularity condition of oClingo.

This paper is structured as follows: in the following section we provide details of oClingo, including its modularity condition. We then introduce the existing concept of Agent Logic Programs, our online extension, and provide a translation to oClingo. In Section 5 we prove that modularity is guaranteed. We conclude with a short discussion.

2 Background

First we recapitulate standard logic programming and answer set programming terminology. Rules are of the form

$$h_r \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n.$$

where each a_i is an atom of the form $p(t_1, \ldots, t_k)$ and each t_i is a term (constant, variable, or function). The *head* h_r of rule r is either an atom, a *cardinality constraint* of the form $l\{h_1, \ldots, h_k\}u$ in which l, u are integers and $h_1, \ldots, h_k$ are atoms, or the special symbol $\bot$. If h_r is a cardinality constraint, we call r a *choice rule*, and an *integrity constraint* if $h_r = \bot$. We denote the atoms occurring in h_r by $head(r)$, ie.

$head(r) = \{h_r\}$ if h_r is an atom, $head(r) = \{h_1, \ldots, h_k\}$ if $h_r = l\{h_1, \ldots, h_k\}u$, and $head(r) = \emptyset$ if $h_r = \bot$. The atoms occurring positively and negatively in the body are denoted by $body(r)^+ = \{a_1, \ldots, a_m\}$ and $body(r)^- = \{a_{m+1}, \ldots, a_n\}$.

A logic program R is a set of rules; $atom(R)$ denotes the set of atoms occurring in R; and $head(R) = \cup_{r \in R} head(r)$ is the collection of all head atoms. The ground program $grd(R)$ is the set of all ground rules constructable from rules $r \in R$ by substituting every variable in r with some element of the Herbrand Universe of R. For further details we recommend several comprehensive works (Baral, 2003; Simons et al., 2002; Gebser et al., 2008).

Before we introduce the modularity condition that is essential to oClingo, we need to define the projection of a rule onto a set of atoms, and how we construct a 'module' from a program. Note that an 'incremental' logic program is constructed from a base module and two parametrized, 'incremental modules' that are instantiated for successive time steps t: a cumulative module $P[t]$ (which is accumulated) and a volatile module $Q[t]$ (which is purged after each step). Definitions 1–4 below are all taken from (Gebser et al., 2011).

Definition 1. *The* projection $P|_X$ *of a program P onto a set X of (ground) atoms is:*

$$\left\{ h_r \leftarrow body(r)^+ \cup L \;\middle|\; \begin{array}{l} r \in P,\ body(r)^+ \subseteq X, \\ L = \{not\ c \mid c \in body(r)^- \cap X\} \end{array} \right\}$$

Informally, the projection accepts rules whose literals are constructed from the projected set X. All positive literals must be represented in X (or the rule is rejected). In contrast, the negative literals that do not belong to X are simply pruned from the rule.

Example Consider the following (ground) logic program

$$\{ p_1 \leftarrow a, b. \quad p_2 \leftarrow a, not\, b, not\, c. \quad p_3 \leftarrow b, c. \}$$

and set $\{a, b\}$ of atoms. The projection preserves the first rule (all atoms are in the set), removes a negation from the second rule (c is missing), and eradicates the last rule (a positive literal is missing from the set):

$$\{p_1 \leftarrow a, b. \;\; p_2 \leftarrow a, not\, b.\}$$

Definition 2. *Generate a* module $\mathbb{P}(I)$ *from a (possibly non-ground) program P and a set of ground atomic inputs I:*

$$\big(grd(P)\big|_Y,\ I,\ head(grd(P)\big|_X)\big)$$

where $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)\big|_X)$. The elements of this tuple are the program, inputs, and outputs, denoted $P(\mathbb{P}), I(\mathbb{P}), O(\mathbb{P})$.

Example Consider the following logic program:

$$\left\{ \begin{array}{c} p_1(Z) \leftarrow a(Z), b(Z). \\ p_2(Z) \leftarrow a(Z), not\, b(Z), not\, c(Z). \\ p_3(Z) \leftarrow b(Z), c(Z). \end{array} \right\}$$

With inputs $I = \{a(1), a(2), b(2), c(2)\}$, the auxiliary sets X, Y are:[1]

$$\begin{aligned} X &= I \cup \{p_1(1), p_1(2), p_2(1), p_2(2), p_3(1), p_3(2)\} \\ Y &= I \cup \{p_1(2), p_2(1), p_2(2), p_3(2)\} \end{aligned}$$

So the inputs to the module are I, and the outputs are those elements in $Y \setminus I$ (displayed above). The ground program itself is:

$$P = \left\{ \begin{array}{c} p_1(2) \leftarrow a(2), b(2). \\ p_2(1) \leftarrow a(1). \\ p_2(2) \leftarrow a(2), \mathit{not}\, b(2), \mathit{not}\, c(2). \\ p_3(2) \leftarrow b(2), c(2). \end{array} \right\}$$

Definition 3. *The* join $\mathbb{P} \sqcup \mathbb{Q}$ *of two modules* $\mathbb{P}$ *and* $\mathbb{Q}$ *is:*

$$\big(P(\mathbb{P}) \cup P(\mathbb{Q}), \quad (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})), \quad O(\mathbb{P}) \cup O(\mathbb{Q}) \big)$$

provided $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ *and there is no strongly connected component in the positive dependency graph of* $P(\mathbb{P}) \cup P(\mathbb{Q})$[2] *that shares atoms with both* $O(\mathbb{P})$ *and* $O(\mathbb{Q})$.

Example Consider the following two modules:

$$\begin{pmatrix} \{p(a) \leftarrow i(a).\ \ p(b) \leftarrow i(b).\}, \\ \{i(a), i(b)\}, \\ \{p(a), p(b)\} \end{pmatrix} \begin{pmatrix} \{q(c) \leftarrow i(c).\}, \\ \{i(c)\}, \\ \{q(c)\} \end{pmatrix}$$

The outputs $\{p(a), p(b)\}$ and $\{q(c)\}$ are disjoint, as are the separate sub-dependency graphs, so the join is well-defined:

$$\left(\left\{ \begin{array}{c} p(a) \leftarrow i(a). \\ p(b) \leftarrow i(b). \\ q(c) \leftarrow i(c). \end{array} \right\}, \left\{ \begin{array}{c} i(a), \\ i(b), \\ i(c) \end{array} \right\}, \left\{ \begin{array}{c} p(a), \\ p(b), \\ q(c) \end{array} \right\} \right)$$

Example The following join is undefined:

$$\begin{pmatrix} \{p \leftarrow q.\}, \\ \{q\}, \\ \{p\} \end{pmatrix} \sqcup \begin{pmatrix} \{q \leftarrow p.\}, \\ \{p\}, \\ \{q\} \end{pmatrix}$$

In this case the combined program has a strongly connected component $\{p, q\}$, which contains outputs from each module.

Finally, an online progression defines the input to each module (cumulative E_i and volatile F_i) at each step (e_i and f_i). We can now reproduce the definition of modularity as presented in (Gebser et al., 2011):

Definition 4. *An online progression* $(E_i[e_i], F_i[f_i])_{i \geq 1}$ *is* modular *with respect to an incremental logic program* $(B, P[t], Q[t])$ *if:*

[1] Recall that grounding yields *all* variable substitutions.

[2] Defined as $(\mathit{atom}(P(\mathbb{P}) \cup P(\mathbb{Q})), \{(a, b) \mid r \in P(\mathbb{P}) \cup P(\mathbb{Q}), a \in \mathit{head}(r),\ b \in \mathit{body}(r)^+\})$.

$$\begin{aligned}
\mathbb{P}_0 &= \mathbb{B}(I_B) = \mathbb{B}(\emptyset) \approx (B, \emptyset, \mathit{head}(B))\\
\mathbb{P}_n &= \mathbb{P}_{n-1} \sqcup \mathbb{P}[t/n]\left(O(\mathbb{P}_{n-1}) \cup I_{P[t/n]}\right)\\
\mathbb{E}_0 &= (\emptyset, \emptyset, \emptyset)\\
\mathbb{E}_n &= \mathbb{E}_{n-1} \sqcup \mathbb{E}_n[e_n]\left(O(\mathbb{P}_{e_n}) \cup O(\mathbb{E}_{n-1}) \cup I_{E_n[e_n]}\right)\\
\mathbb{R}_{j,k} &= \mathbb{P}_k \sqcup \mathbb{E}_j \sqcup \mathbb{Q}[t/k]\left(O(\mathbb{P}_k) \cup I_{Q[t/k]}\right) \sqcup \mathbb{F}_j[f_j]\left(O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I_{F_j[f_j]}\right)
\end{aligned}$$

are well-defined for all $j, k \geq 1$ *such that* $e_1, \ldots, e_j, f_j \leq k$.

Essentially this definition describes how different modules must not produce conflicting outputs. An incremental logic program that satisfies this condition can be efficiently computed stepwise by oClingo. The programs considered below have inputs $I_B = I_{Q[t/k]} = \emptyset$, and $I_{P[t/n]}$ is the set of external actions possible at time $t = n$.

3 Online Agent Logic Programs

The purpose of Agent Logic Programs is to provide high-level control programs for agents using a combination of declarative programming with reasoning about actions (Drescher et al., 2009). The syntax of these programs is kept very simple: standard (definite) logic programs are augmented with just two special predicates, one—written $do(\alpha)$—to denote the execution of an action by the agent, and one—written $?(\phi)$—to verify properties against (the agent's model of) the state of its environment. This model, and how it is affected by actions, is defined in a *separate* action theory. Notably, this clear abstraction improves the elaboration tolerance of agent control (McCarthy, 2007) since independent components are uncoupled and only minor changes are required to adapt to similar domains.

Control of these programs begins with the `main` predicate and continues until it is satisfied, in the style of Prolog-like meta-interpreters, eg. GOLOG. Loops are achieved with recursion.

Example. Consider a simple mail delivery robot within an office environment like the one shown in Fig. 1. Users asynchronously inform the robot of packages to pickup (via exogenous actions). If the robot has tasks, then it must move to pick the requested packages from their requested locations and deliver them. As a further extension, the robot has a limited capacity, so it must schedule pickups and deliveries subject to this hard constraint.

A clean axiomatization of this problem is provided in Fig. 2. The main procedure checks with an underlying action theory whether a request has been made. If so, it calls the `handle` procedure that moves arbitrarily, picks up the package, moves again, and delivers it. We rely on the preconditions of the `pickup` and `drop` actions to guide the system's choice of movement and delivery order, but beyond this the robot has complete freedom.

This means that we can ignore tedious details for managing capacity, path planning, and load balancing—the system will take care of these based on simple declarative descriptions. If we wish to modify the default behavior, we can always introduce a new constraint, or completely override the behavior with new rules.

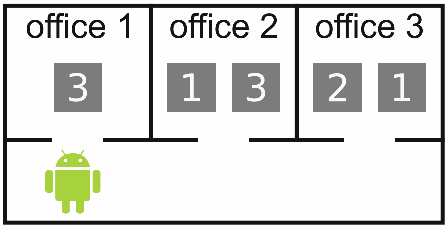

Fig. 1. The mail delivery problem

```
main :- ?(req(F,T)), handle(F,T), main.
main :- ?(noreq).

move.
move :- do(go_up), move.
move :- do(go_down), move.

handle(F,T) :- move, do(pickup(F,T)),
      move, do(drop(F,T)).
```

Fig. 2. An Agent Logic Program for the mail delivery problem

Definition 5 (from (Drescher et al., 2009)). *A logic program (signature Π, without cardinality constraints) is an* Agent Logic Program *(wrt action theory Σ) if:*

- *Terms are in $\Sigma \cup \Pi$.*
- *If p is an n-ary relation symbol in Π and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a program atom.*
- *$do(\alpha)$ is a program atom if α is an action in Σ.*
- *$?(\phi)$ is a program atom if ϕ is a state property in Σ, ie. a formula (represented as a term) based on fluents in Σ.*
- *do and $?$ may not occur in the head of a rule.*
- *Clauses, programs, and queries are defined as usual for definite logic programs.*

The semantics of such a program are defined in terms of its 'temporal extension' (explained in Section 4 below), along with a suitable action theory. The temporal extension is essential in order to take care of the implicit reference to, and changes of, states by $?$ (test) and do (action execution) respectively, and make this information explicit in the encoding. The overall declarative semantics are then the classical logical semantics of this expanded program along with the action theory.

To extend Agent Logic Programs with exogenous actions, the programs must tolerate 'noise' between their own actions. We adopt from (Shapiro and Pagnucco, 2004) the notion of $exoseq$, a relation that detects an uninterrupted sequence of exogenous actions between two time points. This then complements the original action update mechanism from (Drescher et al., 2009). We also require an 'external' definition—a set χ of rules that act as domain predicates on the set of possible external actions.

Definition 6 (Online Agent Logic Program). *An* online agent logic program *(OALP) consists of two sets of rules—an agent logic program P and external definitions χ—and a special time parameter t. Refer to the terms in P whose predicate symbol has no path (in the dependency graph) to "do" or "$?$" as* rigid. *All other rules in P are the* program. *Restrictions:*

- *Rigid rules $R \subseteq P$ may not refer to any predicates outside of R.*
- *P may not mention distinguished predicates*[3] time/1, exoseq/2, exog/1, exec/3, poss/2, occ/2, holds/2, *and must be negation free.*

[3] *time* and *exog* are domain predicates. In Σ, *holds(f,t)* means (relational) fluent f is true at time t; *occ(a,t)* indicates (primitive or exogenous) action a occurred at time t, provided it was *poss(a,t)*. Σ is responsible for generating and constraining *occ*; *exec* is used internally.

- *For every rule in χ, the last argument of the head must be t. All other variables must be bound by domain predicates in the body of the rule, which may only mention atoms in R (this ensures that χ is groundable).*
- *$head(P)$ and $head(\chi)$ must be disjoint.*

In the absence of exogenous actions this definition decays to original Agent Logic Programs, subject to the fact that OALP's target, oClingo, is an ASP system. This means that Definition 6 acts as as both a definition of OALPs and as a filter of 'ASP-compatible' (original) Agent Logic Programs. Every OALP is an Agent Logic Program, but not vice versa; our new OALPs have extra restrictions[4] to facilitate a direct translation to ASP, but beyond that the definitions are equivalent. Hence OALPs are a subset of Agent Logic Programs, but will be our base for an exogenous extension.

Definition 7 (OALP Action Theory). *An OALP-compatible action theory Σ is any* logic program *that satisfies the following conditions:*

- *A state property ϕ is an expression built from the standard logical connectives and fluent terms $F(\boldsymbol{x}) \in \Sigma$. By $holds(\phi, t)$ we denote the formula obtained from property ϕ by replacing every occurrence of a fluent f by $holds(f, t)$. State properties are used by agent logic programs in $?(\phi)$ atoms. They may not mention* poss/2.
- *Action history is embedded in ground $occ(A, t)$ atoms (zero-or-more per time point).*
- *Σ may not mention predicates* do, ?, time/1, exoseq/2, exog/1, *or* exec/3.
- *The first argument to* poss/2, occ/2 *must be an action; to* holds/2 *a fluent. The second argument is a time point.*
- *Terms in Σ may refer to any time points $\leq t$ (ie. past and present) but not $> t$ (future). The last argument to atoms in the head of rules must be t.*

Such action theories may include their own auxiliary rigids. We also leave open the use of action generators (choice rules), constraints, and domain predicates in order to effectively embed the theory into an answer set program. This permits arbitrary concurrency models. An example action theory is provided in Fig. 3.

Definition 8 (OALP progression). *An OALP progression is a progression of sets of atoms constructed from the heads of the external χ rules, where the ith set corresponds to the ith time point (and has t substituted accordingly). It is a special case of an online progression as defined in (Gebser et al., 2011). Namely $e_i = i$ and $F_i = \emptyset$, for all i.*

Example With input directive

```
#external request(F,T,t) : office(F) : office(T)
```

and rigids `office(1)`, `office(2)`, `office(3)`, we have

$$O(\mathbb{E}_n) = \{request(1,1,n),\ request(1,2,n),\ \ldots,\ request(3,3,n)\}$$

Thus two (arbitrary) valid OALP progressions are:

[4] The restrictions are standard for the field, such as requiring domain predicates (for grounding).

```
#base.
office(1). office(2). office(3).
capacity(0). capacity(1). capacity(2).
exog(request(F,T)) :-
   office(F), office(T).
holds(at(1),0).
holds(capacity(2),0).

#cumulative t.
#external request(F,T,t) :
   office(F) : office(T).

{ occ(go_up,t), occ(go_down,t),
  occ(pickup(F,T),t) :
   office(F) : office(T),
  occ(drop(F,T),t) :
   office(F) : office(T) } 1.
{ occ(request(F,T),t) :
   office(F) : office(T) } 1.

poss(drop(F,T),t) :- holds(at(T),t-1),
   holds(carrying(F,T),t-1).
poss(go_down,t) :- holds(at(F),t-1),
   office(F-1).
poss(go_up,t) :- holds(at(F),t-1),
   office(F+1).
poss(pickup(F,T),t) :- holds(at(F),t-1),
   holds(req(F,T),t-1),
   holds(capacity(C),t-1), C>0.

does(t) :- occ(drop(F,T),t).
does(t) :- occ(go_down,t).
does(t) :- occ(go_up,t).
does(t) :- occ(pickup(F,T),t).

:- does(t), occ(request(F,T),t).
:- occ(A,t), not poss(A,t), not exog(A).
:- not request(F,T,t), occ(request(F,T),t).
:- request(F,T,t), not occ(request(F,T),t).
holds(at(F),t) :- holds(at(F),t-1),
   not occ(go_up,t),
   not occ(go_down,t).
holds(at(F+1),t) :- holds(at(F),t-1),
   occ(go_up,t), office(F+1).
holds(at(F-1),t) :- holds(at(F),t-1),
   occ(go_down,t), office(F-1).

holds(carrying(F,T),t) :-
   holds(carrying(F,T),t-1),
   not occ(drop(F,T),t).
holds(carrying(F,T),t) :-
   occ(pickup(F,T),t).

changes(t) :- occ(pickup(F,T),t).
changes(t) :- occ(drop(F,T),t).
holds(capacity(C-1),t) :-
   holds(capacity(C),t-1), capacity(C-1),
   occ(pickup(F,T),t).
holds(capacity(C+1),t) :-
   holds(capacity(C),t-1), capacity(C+1),
   occ(drop(F,T),t).
holds(capacity(C),t) :-
   holds(capacity(C),t-1),
   not changes(t).

holds(noreq,t) :- not hasreq(t).
holds(req(F,T),t) :-
   holds(req(F,T),t-1),
   not occ(pickup(F,T),t).
holds(req(F,T),t) :-
   occ(request(F,T),t).

hasreq(t) :- holds(req(F,T),t).
```

Fig. 3. A simple OALP-compatible action theory to complement the program in Fig. 2

$$\langle\ \{request(1,2,1)\},\{request(3,2,2)\},\emptyset,\emptyset,\{request(1,2,5)\},\ldots\rangle \quad \text{and}$$
$$\langle\ \{request(1,3,1)\},\emptyset,\{request(2,3,3)\},\ldots\rangle$$

The representation of such a progression in an oClingo-compatible format is provided in Fig. 4.

Agent Logic Programs operate much like the formulation of the planning problem in the Situation Calculus: find all times T where the program is satisfied, ie. $\exists T : main(0,T)$. Formerly, execution was not contingent on external input in any way, and so was completely offline. Now the enumeration of time points depends absolutely on the input actions, and so they can be said to be truly online.

4 Translation to oClingo

With our extension of ALPs to online agent logic programs defined, we can now explore the temporal extension that gives these programs a declarative semantics, along the lines of (Drescher et al., 2009).

```
#step 1. request(1,3,1). #endstep.
#step 2. #endstep.
#step 3. request(2,3,3). #endstep.
#stop.
```

Fig. 4. An example OALP progression in oClingo format

Definition 9 (Time-expanded program). *Given an OALP program P, and an unreferenced constant t, call R the set of rigids in P. The* time-expanded program $\mathcal{T}(P,t)$ *contains the set of rigids rules and time-expanded, non-rigid rules. That is: $\mathcal{T}(P,t) = R \cup \{\mathcal{T}(r) \mid r \in P \setminus R\}$, where non-rigid rule r is time-expanded to $\mathcal{T}(r)$ as follows:*

- *Let the time point index $i = 1$.*
- *For each atom b in the left-to-right ordering of the body of r, replace with:*
  - *$holds(\phi, T_i)$ if $b = ?(\phi)$.*
  - *$exec(act, T_i, T_{i+1})$ if $b = do(act)$. Update $i := i + 1$.*
  - *b (leave unchanged) if $b \in R$.*
  - *otherwise $b = r'(y_1, \ldots, y_m) \in P$ is a 'call' to another OALP rule; augment with two time points $r'(y_1, \ldots, y_m, T_i, T_{i+1})$. Update $i := i + 1$.*
- *Prepend the term $exoseq(T_0, T_1)$ to the start of the body.*
- *Note that the rule must now contain references to time points T_0 to T_i inclusive (otherwise it would be rigid). These variables must be distinct from all variables in the original rule r.*
- *Augment the head atom $h_r(x_0, \ldots, x_n)$ with two time points: $h_r(x_1, \ldots, x_n, T_0, T_i)$.*
- *Replace all references to T_i (the last time point variable) with t (including in the head).*

Finally, augment the program with two special rules:

$$exoseq(T,t) \leftarrow exoseq(T,t-1), occ(A,t), exog(A). \qquad \textit{and}$$
$$exec(A,T,t) \leftarrow exoseq(T,t-1), occ(A,t), not\ exog(A).$$

Informally T_i represents the latest time point as one steps through the rule, so checking a condition with ? is done 'now' relative to the left-to-right execution of the program. Performing an action must take its current time point T_i and yield a new, later time point T'—namely the one resulting from performing that action at time T_i. Note that this may be greater than $T_i + 1$ if a sequence of exogenous actions occurs, so it requires a completely new time point, represented by an independent variable. We say that *do* '*maps*' from one time point to a later one. Similarly, a 'function call' (a body atom referring to the head of another program rule) may rely on actions, so it too must map now to a later time point. Domain predicates of the form $time(T_i)$ are added in Definition 10 to help ground the rules.

Example. The complete, translated, mail delivery program from Fig. 2 is given in Fig. 5. The original program maps to lines 1-35. The (volatile) integrity constraint on line 38 forces oClingo to find answer sets where the OALP program `main` is satisfied.

```
#base.
exoseq(0,0).
time(0).

#cumulative t.
main(T1,t) :- exoseq(T1,T2),
      holds(req(F,T),T2),
      handle(F,T,T2,T3),
      main(T3,t), time(T1),
      time(T2), time(T3).
main(T1,t) :- exoseq(T1,t),
      holds(noreq,t), time(T1).

move(T1,t) :- exoseq(T1,T2),
      exec(go_down,T2,T3), move(T3,t),
      time(T1), time(T2), time(T3).
move(T1,t) :- exoseq(T1,T2),
      exec(go_up,T2,T3), move(T3,t),
      time(T1), time(T2), time(T3).
move(T1,t) :- exoseq(T1,t), time(T1).
handle(F,T,T1,t) :- exoseq(T1,T2),
      move(T2,T3),
      exec(pickup(F,T),T3,T4),
      move(T4,T5), exec(drop(F,T),T5,t),
      time(T1), time(T2), time(T3),
      time(T4), time(T5).

exec(A,T,t) :- occ(A,t), not exog(A),
      exoseq(T,t-1), time(T).

exoseq(T,t) :- occ(A,t), exog(A),
      exoseq(T,t-1), time(T).

exoseq(t,t).
time(t).

#volatile t.
:- not main(0,t).
```

Fig. 5. The result of the translation of the mail delivery program to oClingo's native input. This listing excludes the ASP-compatible action theory provided in Fig. 3. Concatenating the two listings produces a runnable ASP description.

Note that in the absence of exogenous actions, $exoseq(T_1, T_2)$ is true iff $T_1 = T_2$, in which case an expanded OALP decays to the original definition[5] of an expanded Agent Logic Program (Drescher et al., 2009):

Proposition 1. *For a logic program constructed from an Agent Logic Program via Definition 9, all answer sets that satisfy $\neg exog(A)$ for all A, will also satisfy*

$$exoseq(T_1, T_2) \leftrightarrow T_1 = T_2.$$

Definition 10 (OALP to ASP construction). *Given a time-expanded Online Agent Logic Program $\mathcal{T}(P, t)$, external action statements χ, and a compatible action theory Σ, $\mathcal{A}(P, \chi, \Sigma, t)$ constructs an incremental logic program (L_B, L_P, L_Q) as follows:*

- *The base section L_B contains all rigids R from $\mathcal{T}(P, t)$, an $exog/1$ relation derived from χ and R, and two facts $time(0) \leftarrow$ and $exoseq(0, 0) \leftarrow$.*
- *The volatile section L_Q contains a single integrity constraint $\leftarrow main(0, t)$, which means that $head(\mathbb{Q}) = \emptyset$.*
- *Add domain predicates $time(T_0), \ldots, time(T_{i-1})$ to the body of each non-rigid rule in the time-expanded program $\mathcal{T}(P, t) \setminus R$—one for each time point variable used in the rule[6]—and call these rules $\mathcal{T}'$.*
- *The cumulative section L_P contains Σ, the expanded rules $\mathcal{T}'$, and two special facts $time(t) \leftarrow$ and $exoseq(t, t) \leftarrow$.*

[5] To be precise, the original definition allows first-order action theories while we target ASP, so restrictions influence the interface $exec/3$ vs $exec/4$.

[6] We do not need the last $time(T_i)$ domain predicate since T_i was substituted with t, which is already ground, and $time(t)$ is true by construction.

We can now reflect back on the difference between original Agent Logic Programs and our new formulation. Definition 6 merely filters the space of all possible (original) Agent Logic Programs to ones that are 'ASP-compatible'. By applying Definition 9 we get a time-expanded program that is equivalent to the original definition (subject to Proposition 1) and is also safe to convert to an oClingo program as in Definition 10.

5 Modularity Condition

We now present the main theoretical result of this paper.

Theorem 1. *For arbitrary OALP (P, χ, t) and action theory Σ, all OALP progressions of χ are* modular *with respect to the construction $\mathcal{A}(P, \chi, \Sigma, t)$.*

That is, applying our construction to an arbitrary Online Agent Logic Program will lead to an incremental logic program that is safe to run on oClingo—the OALP progression is modular with respect to the construction. The key observation here is that modularity, which is defined in terms of the join of specific modules, is only undefined under circumstances involving the outputs of the two joined modules (the full condition is given in Definition 3). A brief proof sketch involves the subsequent points; we cannot provide a complete proof due to space limitations. The crux of our proof is to guarantee that these outputs (and then strongly connected components involving them) are always disjoint. We rely on the fact that our $exoseq$ implementation is monotonic.

Lemma 1. exoseq *is monotonic: every ground instance of* exoseq *maps one time point to an equal or later one.*

We next establish that the incremental definition of $\mathbb{P}$ (that is, the cumulative section of our constructed program) is safe in the sense that progressive joins are well-defined. First we observe that the outputs of successive cumulative sections are disjoint, and then that the strongly connected components satisfy the join criteria.

Lemma 2. *The outputs of two successive modules $\mathbb{P}_{n-1}$ and $\mathbb{P}[t/n](O(\mathbb{P}_{n-1}) \cup I_{P[t/n]})$ are disjoint.*

Lemma 3. *The union of successive modules $\mathbb{P}_{n-1}$ and $\mathbb{P}[t/n](O(\mathbb{P}_{n-1}) \cup I_{P[t/n]})$ has no strongly connected component that violates Definition 3.*

The second recursive definition for a modular incremental logic program is on $\mathbb{E}$ (online input). We now establish that the join of two successive modules is well-defined.

Lemma 4. *The join of any two successive modules $\mathbb{E}_{n-1}$ and $\mathbb{E}_n[e_n](O(\mathbb{P}_{e_n}) \cup O(\mathbb{E}_{n-1}) \cup I_{E_n[e_n]})$ is always well-defined.*

Finally, the proof of Theorem 1 is trivial once we establish that the joins of any $\mathbb{P}$ and $\mathbb{E}$ modules are always well-defined.

Lemma 5. *The join of any two modules $\mathbb{P}$ and $\mathbb{E}$ is always well-defined.*

6 Conclusion

In this paper we have taken two disparate languages—one with a clean formalism and another with an efficient implementation—and provided an automatic translation that preserves the strengths of each. In doing so we have made Agent Logic Programming a viable alternative to previous languages for describing agent control strategies and behaviors. Not only do Agent Logic Programs now have an efficient implementation, they are also easier to write and learn, since they require only two intuitive operators over plain logic programming. They have also been suitably expressive in our own experiments, even allowing nondeterministic operators in the style of GOLOG, though this is the subject of continuing research. This highlights an important result: our translation augments without introducing any new constraints.

More significantly, this clean language guarantees oClingo's modularity condition, so no complicated proofs or semantic checks are required and users can be assured that a syntactically correct program will run. Contrast this with oClingo's native input which currently requires intimate knowledge and appreciation of oClingo's construction and operation. By offering a safe alternative interface, this makes oClingo or its successor Clingo-4 (Gebser et al., 2014), respectively, a more attractive engine for online applications, and we hope this will encourage further research on online answer set systems.

We have also extended Agent Logic Programming to handle online (exogenous) actions, so it may now be applied to genuine cognitive robotics applications; we consider this validation of its expressive powers important future work. This aligns well with the stated goals of oClingo: to effectively reason about real-time dynamic systems running online in changing environments.

The last obvious limitation of Agent Logic Programming—the lack of negation—also needs attention. Previous work in this regard has yielded a complicated, though functional, formulation involving negation-as-failure (Brewka et al., 2012). So far we do not consider negation to be an essential aspect of these programs, though this too is being investigated.

References

Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (2003)

Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)

Brewka, G., Strass, H., Thielscher, M.: Declarative strategies for agents with incomplete knowledge. In: Rosati, R., Woltran, S. (eds.) NMR (2012)

Dastani, M., de Boer, F.S., Dignum, F., Meyer, J.J.C.: Programming agent deliberation: An approach illustrated using the 3APL language. In: AAMAS, pp. 97–104. ACM (2003)

Drescher, C., Schiffel, S., Thielscher, M.: A declarative agent programming language based on action theories. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS (LNAI), vol. 5749, pp. 230–245. Springer, Heidelberg (2009)

Drescher, C., Thielscher, M.: ALPprolog—A new logic programming method for dynamic domains. Theory and Practice of Logic Programming 11(4-5), 451–468 (2011)

Fikes, R., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2(3-4), 189–208 (1971)

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + control: Preliminary report. In: Leuschel, M., Schrijvers, T. (eds.) ICLP Technical Communications. Theory and Practice of Logic Programming, Online Supplement (2014)

Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 54–66. Springer, Heidelberg (2011)

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.) ICLP 2008. LNCS, vol. 5366, pp. 190–205. Springer, Heidelberg (2008)

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

Lloyd, J.W.: Foundations of Logic Programming. Springer (1987)

McCarthy, J.: From here to human-level AI. Artificial Intelligence 171(18), 1174–1182 (2007)

McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press (1969)

Morley, D., Myers, K.: The SPARK agent framework. In: AAMAS, vol. 2, pp. 714–721. IEEE Computer Society (2004)

Mueller, E.T.: Commonsense Reasoning. Morgan Kaufmann (2006)

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

Shapiro, S., Pagnucco, M.: Iterated belief change and exogeneous actions in the situation calculus. In: de Mántaras, R.L., Saitta, L. (eds.) ECAI, pp. 878–882. IOS Press (2004)

Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)

Thielscher, M.: FLUX: A logic programming method for reasoning about agents. Theory and Practice of Logic Programming 5(4-5), 533–565 (2005)