Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 24–33
http://www.floc-conference.org/ICLP-home.html

HEX PROGRAMS WITH ACTION ATOMS

SELEN BASOL [1] AND OZAN ERDEM [1] AND MICHAEL FINK [2] AND GIOVAMBATTISTA IANNI [3]

[1] Faculty of Engineering and Natural Sciences, Sabancı University, Istanbul 34956, Turkey
E-mail address: `{selenbasol,ozanerdem}@su.sabanciuniv.edu`

[2] Institut für Informationssysteme, TU-Wien, Favoritenstraße 9-11, 1040 Wien, Austria
E-mail address: `fink@kr.tuwien.ac.at`

[3] Dipartimento di Matematica, Univ. della Calabria, P.te P. Bucci, Cubo 30B, 87036 Rende, Italy
E-mail address: `ianni@mat.unical.it`

ABSTRACT. HEX programs were originally introduced as a general framework for extending declarative logic programming, under the stable model semantics, with the possibility of bidirectionally accessing external sources of knowledge and/or computation. The original framework, however, does not deal satisfactorily with stateful external environments: the possibility of predictably influencing external environments has thus not yet been considered explicitly. This paper lifts HEX programs to ACTHEX programs: ACTHEX programs introduce the notion of action atoms, which are associated to corresponding functions capable of actually changing the state of external environments. The execution of specific sequences of action atoms can be declaratively programmed. Furthermore, ACTHEX programs allow for selecting preferred actions, building on weights and corresponding cost functions. We introduce syntax and semantics of ACTHEX programs; ACTHEX programs can successfully be exploited as a general purpose language for the declarative implementation of executable specifications, which we illustrate by encodings of knowledge bases updates, action languages, and an agent programming language. A system capable of executing ACTHEX programs has been implemented and is publicly available.

1. Introduction

HEX programs [Eit05], were originally introduced as a general framework for extending declarative logic programming, under the stable model semantics, with the possibility of bidirectionally accessing external sources of knowledge and/or computation. For instance, a rule like

$$pointsTo(X,Y) \leftarrow \&hasHyperlink[X](Y), url(X).$$

1998 ACM Subject Classification: I.2.4 [**Knowledge Representation Formalisms and Methods**]: Representation languages; I.2.3 [**Deduction and Theorem Proving**]: Inference engines, Logic programming, Nonmonotonic reasoning and belief revision; F.4.1 [**Mathematical Logic**]: Computational logic.

Key words and phrases: Answer Set Programming, Logic programming interoperability, Action languages.

This work was partially supported by the Vienna Science and Technology Fund (WWTF) under grant ICT08-020, by the Italian Research Ministry (MIUR) under project INTERLINK II04CG8AGG, and by the Regione Calabria and the EU under POR Calabria FESR 2007-2013 (PIA project of DLVSYSTEM s.r.l.).

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.24

might be devised for obtaining pairs of URLs (X, Y), where X actually links Y on the Web, and $\&hasHyperlink$ is an *external predicate* construct.

The possibility of accessing multiple external sources of knowledge has no significant constraint in HEX programs: in particular, besides constant values, relational knowledge (predicate extensions) can flow from external sources to the logic program at hand and viceversa, and recursion involving external predicates is allowed under reasonable safety assumptions.

It has been illustrated how HEX-programs qualify themselves for actual implementation of action and/or planning languages. As an example, in [Eit05] it is shown how the so called *code call* construct of agent programs as defined in [Eit99] can be embedded in HEX-programs using the notion of external predicate.

As a further example, HEX-programs constitute a generalization of description logic programs as defined in [Eit08]: it is made possible to push additional, hypothetical assertions to an external description logic knowledge base L, and then subsequently query the augmented knowledge base L'. However, it is not possible to push persistent assertions to L: in fact, HEX-programs do not contemplate the possibility of changing the state of external sources. For instance, it can be desirable having a program fragment like

$$new(X) \vee old(X) \leftarrow \&addToFavorites[X], new(X).$$

where intuitively $\&addToFavorites[X]$ is *a)* true for all (and only) the values of X which do not appear in a given external list L of favorite URLs, and *b)* has the side effect of adding X to L if X is not already in L. However, one might wonder what the semantics of a program including the above fragment should be, noticing that $\&addToFavorites$ changes its outcome depending on its state (the list L). Hence, the sequence of state changes due to $\&addFavorites$ would be predictable only if the rule evaluation order in the logic program at hand is operationally specified and known by the programmers.

Updates on external environments changing their state are desired in a variety of contexts, mainly: *1)*, when the actual execution of a plan is expected: in this setting, a change in the environment the agent at hand is acting in is implicitly prescribed; also, the order of execution of plan actions and their effect must be predictable and, indeed, this is the general setting which logic-based action languages are devised to reason about [Gel93]; *2)* when an answer set solver is interfaced with other (stateful) applications: the latter usually elaborate on data depending on answer sets computed, which can be then subsequently exploited for synthetis of new logic programs and evaluation thereof.

In the former case, the logic programming community (and particularly, the nonmonotonic reasoning community), has devoted extensive research towards reasoning about actions and planning, but only a few works (see e.g. [Sub00]) considered the support for actual execution of agent actions explicitly. In the latter case, applications have been developed by the Answer Set Programming community usually resorting to handcrafted solutions, like ad hoc post-parsing of answer sets[1], or developing ad hoc libraries for invoking answer set solvers from other development environments (see, e.g. [Ric03, Pir08]).

Although HEX-programs interface well with external sources of knowledge, it turns out that some structural limitations prevent addressing the issue of having impact on external environments in a satisfactory way: first, external functions associated to external predicates are inherently stateless; second, but more importantly, HEX-programs are fully declarative:

[1]An extensive list of known applications of ASP can be found at
`http://www.kr.tuwien.ac.at/research/projects/WASP/showcase.html`

this implies that when writing an HEX program, it is not predictable whether and in which order an external function will be evaluated.

To this end, we lift HEX programs to ACTHEX programs. ACTHEX programs introduce the notion of *action predicate* and *action atom*. Differently from external predicates, action predicates have impact on external environments and might trigger state changes and side effects. Action predicates are associated to corresponding (executable) functions. The framework allows *a)* to express and infer a predictable order of execution for action atoms, *b)* to express soft (and hard) preferences among a set of possible action atoms, and *c)* to actually execute a set of action atoms according to a predictable schedule. It is worth remarking that ACTHEX programs do not represent an action language in a strict sense. The main goal of the language is *1)* to provide a complementary extension to logic programming over which existing action, planning and agent languages can be grounded, and *2)* to provide a tighter and semantically sound framework for interfacing logic programs with applications of arbitrary nature.

2. Syntax and Semantics

Intuitively, ACTHEX programs extend HEX programs allowing rules like

$$\#robot[move, D]\{b, T\}[2:1] \leftarrow direction(D), time(T).$$

the above can be seen as a rule for scheduling a movement of a given robot in direction D with execution order T. Action atoms are executed according to *execution schedules.* The latter in turn depend on answer sets, which in their generalized form, can contain action atoms. The order of execution within a schedule can be specified using a *precedence* attribute (which in the above rule is set by the variable T); also actions can be associated with weights and priority levels (the values 2 and 1 above, respectively). Action atoms allow to specify whether they have to be executed *bravely* (the b switch above), *cautiously* or *preferred cautiously*, respectively meaning that an action atom a can get executed if it appears in at least one, all, or all *best cost* answer sets. We give next the formal syntax and semantics of the language.

Syntax. Given a finite alphabet Σ, we denote as $\mathcal{C}$, $\mathcal{X}$, $\mathcal{G}$, and $\mathcal{A}$ mutually disjoint subsets of Σ^* whose elements are respectively called constant names, variable names, external predicate names, and action predicate names. Elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ (resp., $\mathcal{A}$) are prefixed with "&" (resp. "#"). Note that names in $\mathcal{C}$ serve both as constant and predicate names.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms.* A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots Y_n)$, where $Y_0, Y_1, \ldots Y_n$ are terms; $n \geq 0$ is the arity of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1 \ldots Y_n)$. The atom is ordinary, if Y_0 is a constant. For example, $(x, type, c)$, $node(X)$, and $D(a, b)$, are atoms; the first two are ordinary atoms. An external atom [Eit05] is of the form $\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$ where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called input and output lists, respectively), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for input and output lists, respectively. An action atom is of the form $\#g\,[Y_1, \ldots Y_n]\,\{o, r\}\,[w : l]$ where $Y_1, \ldots, Y_n$ is a list of terms (called input list), and $\#g$ is an action predicate name. We assume that $\#g$ has fixed length $in(\#g) = n$ for its input list. $o \in \{b, c, c_p\}$ is called the *action option.*

Depending on the value of o, the action atom is called *brave, cautious, preferred cautious*, respectively.

Optional attributes r, w and l range over positive integers and variables[2], and are called *action precedence*, *action weight* and *action level* respectively. For an action atom a, we denote by $pr(a), w(a)$, and $l(a)$ its precedence, weight, and level, respectively. Concerning the latter two, we remark that they are reminiscent of the corresponding attributes of so-called weak constraints, but refrain from further illustration for space reasons.

Example 2.1. The action atom $\#robot[move, left]\{b, 1\}$ may be devised for moving a robot to the left. Here, we have that $in(\#robot) = 2$. This atom features the option b executed with precedence 1, while weight and level information are not given.

A rule r is of the form $\alpha_1 \vee \ldots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\, \beta_{n+1}, \ldots, not\, \beta_m$, where $m, n, k \geq 0$, $m \geq n$, $\alpha_1, \ldots \alpha_k$ are atoms or action atoms, and $\beta_1, \ldots \beta_m$ are either atoms or external atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{not\, \beta_{n+1}, \ldots, not\, \beta_m\}$. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*, and if $B(r) = \emptyset$, and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary*, if it does not contain external or action atoms. An ACTHEX *program* is a finite set P of rules. It is *ordinary*, if all rules are ordinary.

Example 2.2. The following is a valid ACTHEX program:

$$\begin{aligned}
evening &\vee morning.\\
\#robot[turnAlarm, on]\{c, 2\} &\leftarrow evening.\\
\#robot[turnAlarm, off]\{c, 2\} &\leftarrow morning.\\
\#robot[move, all]\{b, 1\} &\leftarrow \&getFuel[](high).\\
\#robot[move, left]\{b, 1\} &\leftarrow \&getFuel[](low).
\end{aligned}$$

Semantics. The semantics of ACTHEX programs generalizes that of HEX-programs given in [Eit05], which in turn generalizes traditional answer-set semantics [Gel91]. In the sequel, let P be an ACTHEX program. We will assume that P acts in a *external environment* E, over which action atoms potentially triggered by P might have some effects. ACTHEX programs can in practice be exploited in a variety of different environments (e.g. a relational database, a file system, or the entire Web): we focus here on the semantics of P, and thus we will make no particular assumption on the nature of E besides assuming it as a finite collection of data structures of unspecified nature and size (to take the most general view, assume E as a finite, arbitrarily large, portion of a Turing machine tape surrounded by blanks on both sides).

The *Herbrand base* of P, denoted HB_P, is the set of all possible ground versions of atoms, external atoms and action atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. The grounding of a rule r, $grnd(r)$, is defined accordingly, and the grounding of program P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{C}, \mathcal{X}, \mathcal{G}$, and $\mathcal{A}$ are implicitly given by P.

Example 2.3. Given $\mathcal{C} = \{edge, arc, d, e, 1, 2\}$, some ground instances of $E(X, c)$ are $edge(d, e)$, $arc(arc, e)$; $\#robot[d, N]\{b, X\}$ has ground instances $\#robot\ [d, e]\{b, 1\}$, $\#robot\ [d, d]\{b, 2\}$.

[2]We assume here that $\mathcal{C}$ contains a finite subset of consecutive integers $S = \{0, \ldots, n_{max}\}$.

An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing (ordinary) atoms and action atoms. We say that I is a *model* of atom (or action atom) $a \in HB_P$, denoted $I \models a$, if $a \in I$. With every external predicate name $\&g \in \mathcal{G}$, we associate an (n+m+1)-ary Boolean function $f_{\&g}$, assigning each tuple $(I, y_1, \ldots, y_n, x_1, , \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in \mathcal{C}$. Similarly, with every action predicate name $\#g \in \mathcal{A}$, we associate a $(n+2)$-ary function $f_{\#g}$ with input $(E, I, y_1, \ldots, y_n)$ and returning a new external environment $E' = f_{\#g}(E, I, y_1, \ldots, y_n)$. Note that functions that are associated with action atoms do not have output lists. We say that $I \subseteq HB_P$ is a model of a ground external atom $a = \&g\,[y_1, \ldots, y_n]\,(x_1, \ldots, x_m)$, denoted $I \models a$, iff $f_{\&g}(I, y_1 \ldots, y_n, x_1, \ldots, x_m) = 1$.

Intuitively, functions associated with external atoms model (stateless) calls to external code and/or external sources of knowledge, as originally defined in [Eit05]. The newly introduced notion here is that of action predicates: action atoms can appear in answer sets or not depending on whether they are a consequence of the program at hand or not; functions associated with action predicates serve the purpose of modelling the actual execution of entailed action atoms, i.e., the respective changes on E.

Example 2.4. We associate with $\&reach$ a function $f_{\&reach}$, s.t. $f_{\&reach}(I, G, A, B) = 1$ iff node B is reachable from node A in the graph encoded by means of the binary predicate G . Let $I = \{e(b, c), e(c, d)\}$. Then, I is a model of $\&reach[e, b](d)$, since $f_{\&reach}(I, e, b, d) = 1$. Also, let us associate with $\#insert$ a function $f_{\#insert}$, and assume that E contains an encoding of a knowledge base K expressed as a set of facts. When action atom $\#insert\,[edge, arc]\,\{b, 1\}$ needs to be executed, then the function $f_{\#insert}$ is called with inputs $(E, I, edge, arc)$, for an interpretation I. Intuitively, $\#insert$ might correspond to the act of adding to the extension of the predicate *edge* in K the extension of the predicate *arc* in I.

Let r be a ground rule. We define (i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$, (ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$. We say that I is a *model* of an ACTHEX program P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. We call P *satisfiable*, if it has some model. Given an ACTHEX program P, the *FLP-reduct* of P with respect to $I \subseteq HB_P$, denoted fP^I , is the set of all $r \in grnd(P)$ such that $I \models B(r)$. $I \subseteq HB_P$ is an *answer set* of P iff I is a minimal model of fP^I.

Note that we inherit from the framework of HEX programs the adoption of the notion of reduct as defined by [Fab04] (referred to as *FLP-reduct* henceforth). The FLP-reduct is equivalent to the traditional Gelfond-Lifschitz reduct for ordinary programs, and in our context ensures answer-set minimality, even in the presence of external atoms (see [Eit05] for details). Let $\mathcal{AS}(P)$ be the collection of all the answer sets of program P; the set of *best models* $\mathcal{BM}(P)$ contains the answer sets of P minimizing an objective function H_P. $H_P(A)$ intuitively weighs an answer set A depending on the weights (and levels) of action atoms which are contained in A[3].

Let a be an action atom of the form $\#g\,[y_1, \ldots y_n]\,\{o, r\}$, and $A \in \mathcal{AS}(P)$; a is said to be *executable* in A, if *i)* a is brave (i.e., $o = b$) and $a \in A$, or *ii)* a is cautious (i.e., $o = c$) and $a \in B$ for every $B \in \mathcal{AS}(P)$, or *iii)* a is preferred cautious (i.e., $o = c_p$) and $a \in B$ for every $B \in \mathcal{BM}(P)$. Roughly speaking, once an answer set A is chosen as the one to be

[3]For space reasons, the reader can find the definition of H_P at
`http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin/preferences.html`

executed, action atoms to be executed are selected depending on their action option. Note that, in this respect, the notion of *brave executability* depends on the answer set at hand and thus slightly differs from the traditional notion of brave entailment.

Given an answer set $A \in \mathcal{BM}(P)$, an *execution schedule* $E_{A,P} = [a_1, \dots, a_n]$ is an ordered list containing all the action atoms executable in A, such that $i < j$ if $pr(a_i) < pr(a_j)$, for each pair of atoms a_i, a_j appearing in $E_{A,P}$.

Intuitively, an execution schedule for a program gives an order for the action execution compatible with the precedences specified in the program. Note that for action atoms with the same precedence the execution order is not specified.

Given an execution schedule $E_{A,P} = [a_1, \dots, a_n]$, let $E_0 = E$, and for $i > 0$, $E_i = f_{a_i}(E_{i-1}, A, y_1, \dots, y_m)$. We define $EX(E_{A,P}) = E_n$ as the *execution outcome* of $E_{A,P}$, and $\mathcal{EX}(P) = \{E_{A,P} \mid A \in \mathcal{BM}(P)\}$.

In general, given a program P, we consider $\mathcal{AS}(P)$, $\mathcal{BM}(P)$ and $\mathcal{EX}(P)$ as different facets of the semantics of P. In particular, the *execution outcome* of P is $EX(E_{A,P})$ for an execution schedule $E_{A,P} \in \mathcal{EX}(P)$ of choice. We simply assume that a deterministic rule for choosing $E_{A,P}$ is given[4].

Example 2.5. Let A_1, A_2, A_3 be three answer sets of a given program $P_{ex2.5}$, where A_1, $A_2 \in \mathcal{BM}(P_{ex2.5})$. Let $a_1 = \#insert\,[e, g_1]\,\{b, 1\}$, $a_2 = \#insert\,[e, g_2]\,\{c, 5\}$, $a_3 = \#insert\,[e, g_3]\,\{c, 2\}$, $a_4 = \#insert\,[e, g_4]\,\{c_p, 2\}$, $a_5 = \#insert\,[e, g_5]\,\{b, 1\}$, and let $A_1 = \{a_1, a_2, a_3, a_4, a_5\}$, $A_2 = \{a_2, a_4\}$, $A_3 = \{a_2, a_5\}$.

Since $A_3 \notin \mathcal{BM}(P_{ex5})$, possible choices of answer sets are A_1 and A_2. If we choose A_1, brave atoms a_1, a_5, cautious atom a_2 and preferred cautious atom a_4 are executable since $a_1, a_5 \in A_1$, where a_2 appears in all the answer sets and a_4 appears in both A_1 and A_2 . A_1 has two possible execution schedules which are $[a_1, a_5, a_4, a_2]$, and $[a_5, a_1, a_4, a_2]$.

For the case that A_2 is selected, cautious atom a_2 and preferred cautious atom a_4 are executable since a_2 appears in all answer sets, and a_4 appears in A_1 and A_2. Thus, the only possible execution schedule for A_2 is $[a_4, a_2]$.

3. Applications of ACTHEX programs

In this section, we provide evidence for the versatility of ACTHEX by discussing several application scenarios, including encodings of existing action-based KR formalisms.

Action languages. We use action language $\mathcal{C}$ [Giu98] as a representative for sketching how action languages can be reduced to ACTHEX programs. The relationship to logic programming is well-known: we follow a transformation from [Lif99].

The semantics of $\mathcal{C}$ is defined in terms of transition diagrams which put in relationship propositional *action* and *fluent* atoms. The possible state evolution specified in transition diagrams can equivalently be characterized as a logic program expressed in terms of predicates having a time attribute, which are used for encoding truth values of different action and fluent variables at different times. Not surprisingly, the precedence attribute of action atoms can intuitively capture the notion of time as in [Lif99].

Consider causal laws defined as either a *static law* of the form "**caused** F **if** $L_1 \wedge \cdots \wedge L_m$", or a *dynamic law* of the form "**caused** F **if** $L_1 \wedge \cdots \wedge L_m$ **after** $L_{m+1} \wedge \cdots L_n \wedge L_{n+1} \wedge$

[4]For the sake of efficiency, our implementation executes the first execution schedule obtained from the first computed answer set: other selection criteria are of course possible.

$\cdots \wedge L_k$", where F is a fluent literal, L_i is a fluent literal for $1 \leq i \leq n$, and respectively an action name for $n+1 \leq i \leq k$. An *action description* is a set of causal laws.

Given an action description D and a *maximum time* t, following [Lif99], a dynamic law $l \in D$ of the form above can be translated to the ordinary rule $F'(T+1) \leftarrow$ *not* $\overline{L}'_1(T+1), \ldots$, *not* $\overline{L}'_m(T+1), L'_{m+1}(T), \ldots, L'_k(T)$, where F' is a unary predicate associated to fluent F, while L'_i, $\overline{L}'_i$ are unary predicates associated to fluents L_i, $1 \leq i \leq n$, respectively to actions L_i, $n+1 \leq i \leq k$, and their complements[5]. We then put in connection action atoms with actions by means of rules $\#L_i\{o,T\} \leftarrow L_i(T).$, $n+1 \leq i \leq k$, where $\#L_i$ is a newly introduced action atom which is responsible of executing the action L_i, and o is an action option. By adding other auxiliary rules (e.g. guessing rules $b(T) \vee \overline{b}(T) \leftarrow T \leq t$ for each action b), and setting $o = b$, we obtain a program P_D whose execution schedules $\mathcal{EX}(P_D)$ correspond to so-called *histories* (paths) of length t in D. An execution plan $e \in \mathcal{EX}(P_D)$ can then be materially executed. Similarly, preference orderings between actions as in the language $\mathcal{P}P$ and variants thereof [Son06], can be attached to action atoms: for an ordering $L_1 < \cdots < L_n$ among actions one can introduce corresponding integer weights $w_1 < \cdots < w_n$ and rules $\#L_i\{O,T\}[w_i : 1] \leftarrow L_i(T)$.

Knowledge Base Updates. As another potential usage of ACTHEX programs, we mention the possibility of updating knowledge bases, e.g., as achieved by the predicates *assert* and *retract* in Prolog. We assume that external environments contain a collection C of knowledge bases accessible by names, and consider abstract action constructs $assert(kb, f)$ and $retract(kb, f)$, which respectively should add or remove a statement f from a given knowledge base kb. The above can be grounded to ACTHEX programs, introducing action predicates $\#assert_k$ and $\#retract_k$, for $k > 0$[6]. An atom $\#assert_k[kb, a_1, \ldots, a_k]\{o,p\}$, (resp. $\#retract_k[kb, a_1, \ldots, a_k]\{o,p\}$) adds to (resp. removes from) the knowledge base kb the assertion $a_1|\ldots|a_k$, for $a_i|a_j$, being the string concatenation of a_i and a_j.

For instance, the rule $\#assert_3[kb, \text{“}n(\text{”}, X, \text{“}).\text{”}]\{b,1\} \leftarrow node(X)$. encodes the possible addition of facts $n(c)$ for each c such that $node(c) \in A$, for an answer set A. The above constructs can be fruitfully combined with reasoning over the given knowledge bases: to this end, we introduce the action atom $\#execute[kb]\{o,p\}$. Assuming the kb is a valid ACTHEX program, when such an atom belongs to the current execution schedule, it gets executed by evaluating kb and the resulting execution schedule. Note that whether $\#assert$, $\#retract$ and $\#execute$ actions will be executed depend on reasoning on the program at hand: this opens a variety of possibilities, e.g. belief revisions, and, in general, observe-think-act cycles [Kow99][7]. Note that the evaluation of programs with this kind of construct might not terminate in general: this issue is subject of ongoing study.

Translation of Agent Programs. Agent programs can also be realized in the ACTHEX framework. We consider logic-based *agent programs* as developed in [Sub00], consisting of rules of

[5]We can assume a constraint $\leftarrow L'_i(T), \overline{L}'_i(T)$ is added for each L_i. Note that the current implementation of ACTHEX programs allows for strong negation, by which an atom $\overline{L}'(T)$ can be conveniently modelled as $\neg L'(T)$.

[6]Our implementation of ACTHEX programs conveniently allows to program and group families of action atoms, like the above, using variable length parameter lists.

[7]An example ACTHEX program containing update actions is given at
`http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin/actionplugin_example1.html`

the form $Op_0\alpha_0 \leftarrow \chi, [\neg]Op_1\alpha_1, \ldots, [\neg]Op_m\alpha_m$, governing an agent's behaviour. The Op_i are *deontic modalities*, the α_i are *action status atoms*, and χ is a *code-call condition*.

For instance, $Do\ dial(N) \leftarrow in(N, phone(P)),\ O\ call(P)$, intuitively states that the agent should dial phone number N if she is obliged to call P. In [Sub00], a translation of an agent program $AG(P)$ to a logic program P is given, such that the answer sets of P correspond to the so-called *reasonable status sets* of $AG(P)$. We build on this transformation and model code-call conditions (which, e.g., provide access to actual sensor readings) using external atoms as already described in [Eit05]. Similarly, we model Do atoms as action atoms in our framework using rules of the sort $\#action_\alpha[\ldots]\{b\} \leftarrow Do\ \alpha$. A framework implementing this translation is available[8], featuring *a)* the translation of agent programs to ACTHEX programs, *b)* incorporating the actual execution of *Do*-able actions and *c)* an implementation of message box facilities for agents.

Other applications. ACTHEX programs can be exploited in a variety of other contexts, ranging from database access to interaction with actual web sources. We developed an example[9] illustrating how to exploit reasoning in ASP for choosing meeting schedules of two teams. Events are extracted from actual Google Calendars[10] of two teams; meeting dates are selected using ASP reasoning; eventually, the chosen events are posted to the calendars of the teams using an action atom of the form

$\#createEvent[Team, Url, \text{``}ActHexMeeting\text{''}, Date, User, Password]\{b, 1\}$.

4. Implementation Notes

An implementation of ACTHEX programs has been realized and is available[11] as an extension to the dlvhex system[12]. With respect to the traditional workflow of an answer set solver, the system computes execution schedules and executes one of it according to: *i)* the semantics of ACTHEX programs, *ii)* the selection policy of execution schedules described in Section 2, and *iii)* the associated executable functions provided for action predicates. The system is equipped with a toolkit enabling users to develop their own libraries of action predicates: some example libraries are available. In particular, the KBModaddon library constitutes a generalization of update action atoms as shown in Section 3 (it is, e.g., possible to execute arbitrary command line statements, and to assert and retract arbitrary statements from knowledge bases). An example library allowing access and modification to Google Calendars is also publicly available.

5. Related Work and Conclusions

Our work has points of contact with some lines of research which can be grouped as follows. *Action languages* serve the purpose of providing a declarative language for specifying causal theories [Giu98, McC97], allowing to assert not only the truth of a proposition, but also that there is a cause for it to be true. In this respect, they provide a formalism for the declarative representation of dynamic domains and gave rise to logic-based planning

[8] `http://students.sabanciuniv.edu/~ozanerdem/AgentToHex.html`

[9] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin/actionplugin_example2.html`

[10] `http://www.google.com/calendar`

[11] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin.html`

[12] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

systems such as CCLAC [Giu04] and DLV$^{\mathcal{K}}$ [Eit03]. The two systems mentioned are based on transformations [Lif99, Gel93] to logic programming under the answer set semantics, however other (nonmonotonic) reasoning engines can be exploited for causal reasoning in action domains as well (cf, e.g., [Tur96, Kak01, Lin00]).

ACTHEX programs generalize HEX programs which in turn generalize ASP programs, and thus can be similarly used to implement planning systems based on action languages (as shown in Section 3). When resorting to ACTHEX, however, action atoms also encode their actual execution, enabling a variety of applications. For instance, this allows for interweaving plan generation and action execution seamlessly within a coherent declarative framework, which may, e.g., be utilized for an integrated approach to monitoring plan execution. For instance, [Nie07] extends the action language $\mathcal{K}$ towards conditional planning: building on HEX programs, they introduce external function calls in causal rules to import fluent information from an external source. The introduction of action atoms makes it possible to extend the framework coping with action execution and monitoring their success.

Logic-based agent programming constitutes a further natural application domain for ACTHEX programs: intelligent agents require reasoning and/or planning capabilities for acting in dynamic environments, and using logic programming for the declarative specification of a respective observe-think-act cycle [Kow99] is a reasonable choice. ACTHEX may serve as an implementation layer for agent systems built according to this paradigm. We exemplified its suitability providing a transformation of IMPACT agent programs [Sub00] into corresponding ACTHEX programs.

The evaluation of IMPACT agent programs is restricted to stratified negation in its current implementation: the given ACTHEX encoding does not require such a restriction and can handle general agent programs as formally conceived. Similarly, compared to ACTHEX, agent-oriented logic programming languages based on Horn clause languages (e.g., DALI [Cos04], or ALP [Dre09]) lack a declarative concept of negation, which is important from an expressive and practical modelling point of view, for instance to express exceptions. On the other hand, most nonmonotonic logic programming based approaches to agent-oriented programming, (e.g.[Alf06, Alf08, Nie06, Vos05, Lei01]), detach the reasoning process from the actual execution of an agent's actions (which often are termed 'external') and only their (expected) effects are taken into account for further deliberation. For such agent frameworks, ACTHEX can provide the platform for an integrated implementation. In conclusion, ACTHEX is a declarative logic programming framework including a representation for actions that are executed and have an impact on an external environment. Formal properties of the language and further extensions (e.g. parallel execution schedules) are subject to ongoing work. Corresponding results, as well as a more rigorous treatment of the given encodings, will be subject of follow-up work and/or an extended version of this paper.

References

[Alf06] J. J. Alferes, F. Banti, and A. Brogi. An event-condition-action logic programming language. In *JELIA*, pp. 29–42. 2006.

[Alf08] J. J. Alferes, A. Gabaldon, and J. Leite. Evolving logic programming based agents with temporal operators. In *IAT*, pp. 238–244. 2008.

[Cos04] S. Costantini and A. Tocchio. The DALI logic programming agent-oriented language. In *JELIA*, pp. 685–688. 2004.

[Dre09] C. Drescher, S. Schiffel, and M. Thielscher. A declarative agent programming language based on action theories. In *FroCos*, pp. 230–245. 2009.

[Eit99] T. Eiter, V. S. Subrahmanian, and G. Pick. Heterogeneous active agents, I: semantics. *Artif. Intell.*, 108(1-2):179–255, 1999. doi:http://dx.doi.org/10.1016/S0004-3702(99)00005-3.

[Eit03] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system. *Artif. Intell.*, 144(1-2):157–211, 2003.

[Eit05] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI*, pp. 90–96. 2005.

[Eit08] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artif. Intell.*, 172(12-13):1495–1539, 2008.

[Fab04] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *JELIA*, pp. 200–212. 2004.

[Gel91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[Gel93] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *JLP*, 17:301–322, 1993.

[Giu98] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI/IAAI*, pp. 623–630. 1998.

[Giu04] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *AI*, 153(1-2):49–104, 2004. doi:http://dx.doi.org/10.1016/j.artint.2002.12.001.

[Kak01] A. C. Kakas, R. Miller, and F. Toni. E-RES: Reasoning about actions, events and observations. In *LPNMR*, pp. 254–266. 2001.

[Kow99] R. A. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Ann. Math. Artif. Intell.*, 25(3-4):391–419, 1999.

[Lei01] J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. In *ATAL*, pp. 141–157. 2001.

[Lif99] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *LPNMR*, pp. 92–106. 1999.

[Lin00] F. Lin. From causal theories to successor state axioms and strips-like systems. In *AAAI/IAAI*, pp. 786–791. 2000.

[McC97] N. McCain and H. Turner. Causal theories of action and change. In *AAAI/IAAI*, pp. 460–465. 1997.

[Nie06] D. Van Nieuwenborgh, M. De Vos, S. Heymans, and D. Vermeir. Hierarchical decision making in multi-agent systems using answer set programming. In *CLIMA VII*, pp. 20–40. 2006.

[Nie07] D. Van Nieuwenborgh, T. Eiter, and D. Vermeir. Conditional planning with external functions. In *LPNMR*, pp. 214–227. 2007.

[Pir08] G. Pirrotta and A. Provetti. A Java wrapper for answer set programming inferential engines. In *CILC 2008*.

[Ric03] F. Ricca. The DLV Java wrapper. In *APPIA-GULP-PRODE*, pp. 263–274. 2003.

[Son06] T. C. Son and E. Pontelli. Planning with preferences using logic programming. *TPLP*, 6(5):559–607, 2006.

[Sub00] V. S. Subrahmanian, P. A. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. B. Ross. *Heterogenous Active Agents*. MIT Press, 2000.

[Tur96] H. Turner. Representing actions in default logic: A situation calculus approach. In *In Proceedings of the Symposium in honor of Michael Gelfond's 50th birthday (also in Common Sense 96)*. 1996.

[Vos05] M. De Vos, T. Crick, J. A. Padget, M. Brain, O. Cliffe, and J. Needham. LAIMA: A multi-agent platform using ordered choice logic programming. In *DALT*, pp. 72–88. 2005.