HEX Programs with Action Atoms

Selen Basol, Michael Fink, Ozan Erdem, Giovambattista Ianni

Sabancı University (e-mail: selenbasol@sabanciuniv.edu)
Vienna University of Technology (e-mail: fink@kr.tuwien.ac.at)
Sabancı University (e-mail: ozanerdem@sabanciuniv.edu)
Università della Calabria (e-mail: 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 (Eiter et al. 2005) 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. It has been illustrated how HEX-programs qualify themselves for actual implementation of action and/or planning languages. As an example, in (Eiter et al. 2005) it is shown how the so called code call construct of agent programs as defined in (Eiter et al. 1999) can be embedded in HEX-programs using the notion of external predicate. The possibility of accessing multiple external sources of knowledge has no significant constraint in HEX programs: in particular, relational knowledge can flow from external sources to the logic program at hand and viceversa.

As an example, HEX-programs constitute a generalization of description logic programs as defined in (Eiter et al. 2008): it is made possible to push additional, hypothetical assertions to an external description logic knowledge base $L$, and then subjunctively query the augmented knowledge base $L'$. However, it is not possible to push persistent assertions to $L$: in general, HEX-programs do not contemplate the possibility of changing the state of external sources.
This is 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; indeed, this is the general setting which logic-based action languages are devised to reason about (Gelf. and Lif. 1993); and 2) when an answer set solver is interfaced with other applications: the latter usually elaborate on data depending on answer sets computed.

In the former case the logic programming community (and particularly, the non-monotonic reasoning community), has devoted extensive research towards reasoning about actions and planning, but only a few works (see e.g. (Subrahmanian et al. 2000)) 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. (Ricca 2003; Pirrotta and Provetti 2008)).

It turns out that some structural limitations of HEX-programs prevent addressing this issue in a satisfactory way: first, external functions associated to external predicates are inherently stateless. Second, but more importantly, HEX-programs are fully declarative: 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. 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

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

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). Action atoms allow to specify whether they have to be executed bravely

\(^1\) An extensive list of known applications of ASP can be found at http://www.kr.tuwien.ac.at/research/projects/WASP/showcase.html
Given a finite alphabet $\Sigma$, we denote as $r$ the action atom $A_\beta Y_0 B$. Depending on the value of $r$, called fixed length

Elements from $\mathcal{C}$ 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.

Intuitively, a fact with its precedence, weight, and level, respectively. The set of action atoms featuring explicit weight and level values are denoted by $\text{AA}_w(P)$.

Example 1 The action atom $\#\text{robot}[\text{move}, \text{left}][\{b, 1\}]$ may be devised for moving a robot to the left. Here, we have that $\text{in}(\#\text{robot}) = 2$. 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 $\alpha_1 \vee \ldots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, \text{not } \beta_{n+1}, \ldots, \text{not } \beta_m$, 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 $B^-(r) = \{\text{not } \beta_{n+1}, \ldots, \text{not } \beta_m\}$. 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.

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

---

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

---
Example 2 The following is a valid ActHex program:

\[
evening \lor \text{morning},
\]  
\[
\#\text{robot\{turnAlarm, on\}[c, 2]} \leftarrow \text{evening}.
\]  
\[
\#\text{robot\{turnAlarm, off\}[c, 2]} \leftarrow \text{morning}.
\]  
\[
\#\text{robot\{move, all\}[b, 1]} \leftarrow \&\text{getFuel}[[\text{high}]].
\]  
\[
\#\text{robot\{move, left\}[b, 1]} \leftarrow \&\text{getFuel}[[\text{low}]].
\]

Semantics. The semantics of ActHex programs generalizes that of Hex-programs given in (Eiter et al. 2005), which in turn generalizes traditional answer-set semantics (Gelf. and Lif. 1991). 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 \( \text{HB}_P \), 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 \( C \). The grounding of a rule \( r \), \( \text{grnd}(r) \), is defined accordingly, and the grounding of program \( P \) is given by \( \text{grnd}(P) = \bigcup_{r \in P} \text{grnd}(r) \). Unless specified otherwise, \( C, \mathcal{X}, \mathcal{G} \), and \( A \) are implicitly given by \( P \).

Example 3 Given \( C = \{\text{edge, arc, d, e, 1, 2}\} \), some ground instances of \( E(X, c) \) are \( \text{edge}(d, e) \), \( \text{arc}(d, e) \); \( \#\text{robot\{d, N\}[b, X] \} \) has ground instances \( \#\text{robot \{d, e\}[b, 1], \#\text{robot \{d, d\}[b, 2]}. \)

An interpretation relative to \( P \) is any subset \( I \subseteq \text{HB}_P \) containing atoms and action atoms . We say that \( I \) is a model of atom \( a \in \text{HB}_P \), 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 \( n = \text{in}(\&g) \), \( m = \text{out}(\&g) \). With every action predicate name \( \#g \in A \), 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 \( I \subseteq \text{HB}_P \) is a model of a ground external atom \( a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m) \), denoted \( I \models a \), if \( f_{\#g}(I, y_1, \ldots, y_n, x_1, \ldots, x_m) = 1 \).

Intuitively, functions associated with external atoms model (stateless) calls to external code and/or external sources of knowledge, as originally defined in (Eiter et al. 2005). The newly introduced notion here is that of action predicates: functions associated with action predicates serve the purpose of modelling the actual execution of operations influencing \( E \).

Example 4 We associate with \( \&\text{reach} \) a function \( f_{\&\text{reach}} \), st. \( f_{\&\text{reach}}(I, G, A, B) = 1 \) iff \( B \) is reachable in the graph \( G \) from \( A \). Let \( I = \{c(b, c), c(c, d)\} \). Then, \( I \) is
a model of $\kerch[e, b, d]$, since $f_{\kerch}(I, e, b, d) = 1$. Also, let us associate with $\#\text{insert}$ a function $f_{\#\text{insert}}$, and assume that $E$ contains an encoding of a knowledge base $K$ expressed as a set of facts. When action atom $\#\text{insert} [\text{edge, arc}] \{b, 1\}$ needs to be executed, then the function $f_{\#\text{insert}}$ is called with inputs $(E, I, \text{edge, arc})$, for an interpretation $I$. Intuitively, $\#\text{insert}$ might correspond to the act of adding to the extension of the predicate $\text{edge}$ in $K$ the extension of the predicate $\text{arc}$ 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 (Faber et al. 2004) (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 (Eiter et al. 2005) 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 the objective function $H_P$ as defined in Appendix A. $H_P(A)$ intuitively weighs an answer set $A$ depending on the weights of action atoms which are contained in $A$.

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 executed, action atoms to be executed are selected depending on their action option. Note that, in this respect, the notion of brave executability differs from the traditional notion of brave entailment.

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

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

\footnote{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.}
Example 5 Let $A_1$, $A_2$, $A_3$ be three answer sets of a given program $P_{c55}$, where $A_1$, $A_2 \in \text{BM}(P_{c55})$. Let $a_1 = \#\text{insert}[e, g_1] \{b, 1\}$, $a_2 = \#\text{insert}[e, g_2] \{c, 5\}$, $a_3 = \#\text{insert}[e, g_3] \{c, 2\}$, $a_4 = \#\text{insert}[e, g_4] \{e, 2\}$, $a_5 = \#\text{insert}[e, g_5] \{b, 1\}$, 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 $A_3 \not\in \text{BM}(P_{c55})$, 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 $C$ (Giunchiglia and Lifschitz 1998) as a representative for illustrating how action languages can be reduced to acthex programs. The relationship to logic programming is well-known: we follow a transformation from (Lifschitz and Turner 1999).

The semantics of $C$ 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 (Lifschitz and Turner 1999). Consider causal laws defined as either a static law of the form “caused $F$ if $L_1 \land \cdots \land L_m$”, or a dynamic law of the form “caused $F$ if $L_1 \land \cdots \land L_m$ after $L_{m+1} \land \cdots \land L_n \land L_{n+1} \land \cdots \land L_k$”, 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 (Lifschitz and Turner 1999), a dynamic law $l \in D$ of the form above can be translated to the ordinary rule $F'(T+1) \leftarrow \text{not } T_i(T+1), \ldots, \text{not } T_m(T+1)L_{i+1}(T)\ldots L_k(T)$, where $F'$ is a unary predicate associated to fluent $F$, while $T_i, T_m$ 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.$^4$ 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 $b(T) \lor \overline{b}(T) \leftarrow T \leq t$)

---

$^4$ We can assume a constraint $\leftarrow L_i(T), T_i(T)$ is added for each $L_i$. Note that the current implementation of acthex programs allows for strong negation, by which an atom $T_i(T)$ can be conveniently modelled as $\neg L_i(T)$. 

6
for each action \( b \), and setting \( o = b \), we obtain a program \( P_D \) whose execution schedules \( \mathcal{E}(P_D) \) correspond to so-called histories (paths) of length \( t \) in \( D \). An execution plan \( e \in \mathcal{E}(P_D) \) can then be materially executed. Similarly, preference orderings between actions as in the language \( \mathcal{PP} \) and variants thereof (Son and Pontelli 2006), 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 \#\( L_i\{O,T\}[w_i : 1] \leftrightarrow L_i(T) \).

**Knowledge Base Updates.** As another potential usage of acthex programs, we mention the possibility of permanently 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 \( k > 0 \). 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 \( a_1 \cdots \) \( a_k \), for \( a_i \) \( a_j \), being the string concatenation of \( a_i \) and \( a_j \).

For instance, the rule #assert\(_3\)[kb, “n(", X, ")”]\{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 subsequent execution schedule. The above constructs open a variety of possibilities, e.g. belief revisions, and, in general, observe-think-act cycles (Kowalski and Sadri 1999). A sample acthex program using update actions is given in Appendix B.

Note that evaluation of languages 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 (Subrahmanian et al. 2000), consisting of rules of 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 \( \alpha_i \) are action status atoms, and \( \chi \) is a code-call condition. E.g., Do dial(N) \( \leftarrow \) int(N, phone(P)), O call(P), intuitively states that the agent should dial phone number N if she is obliged to call P. In (Subrahmanian et al. 2000), 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 (Eiter et al. 2005). Similarly, we model Do atoms as action atoms in our framework using rules of the sort

\[ \#execute[kb]\{o, p\} \leftrightarrow \text{node}(X). \]

5 Our implementation of acthex programs conveniently allows to program and group families of action atoms like the above using variable length parameter lists.
A framework implementing this translation is available, 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. The example in Appendix C shows how to exploit reasoning in ASP for choosing meeting schedules of two teams. Events are extracted from actual Google Calendars 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, “ActHexMeeting”, Date, User, Password]b1.

4 Implementation Notes
An implementation of ACTHEX programs has been realized and is available as 
an extension to the dlvhex system. 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 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 (Giunchiglia and Lifschitz 1998; McCain and Turner 1997), 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 systems such as CCLAC (Giunchiglia et al. 2004) and DLVX (Eiter et al. 2003). The two systems mentioned are based on transformations (Lifschitz and Turner 1999; Gelf and Lif. 1993) 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., (Turner 1996; Kakas et al. 2001; Lin 2000)).

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,

---

6 http://students.sabanciuniv.edu/~ozanerdem/AgentToHex.html
7 http://www.google.com/calendar
8 http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin.html
9 http://www.kr.tuwien.ac.at/research/systems/dlvhex/
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, Nieuwenborgh et al. (2007) extend the action language $K$ 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 (Kowalski and Sadri 1999) 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 (Subrahmanian et al. 2000) 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 (Costantini and Tocchio 2004), or ALP (Drescher et al. 2009)) 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., (Alferes et al. 2006, 2008, Nieuwenborgh et al. 2006; De Vos et al. 2005; Leite et al. 2001), 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. Properties of the language and further extensions (e.g. parallel execution schedules) are subject to ongoing work.

References


Appendix A The objective function \( H_P(A) \)

We report here the definition of the objective function \( H_P(A) \) (generalized from (Leone et al. 2006)). For an answer set \( A \), \( H_P(A) \) is defined in terms of an auxiliary function \( f_p \) as follows:

\[
\begin{align*}
 f_P(1) &= 1, \\
 f_P(n) &= f_P(n-1) \cdot (|AA_w(grnd(P))|) \cdot w_{\text{max}}^P + 1, \quad n > 1 \\
 H_P(A) &= \sum_{i=1}^{l_{\text{max}}^P} (f_P(i) \cdot \sum_{a \in M_i^P(A)} \text{weight}(a)) 
\end{align*}
\]

where \( w_{\text{max}}^P \) and \( l_{\text{max}}^P \) denote the maximum weight and maximum level over weighted action atoms in \( grnd(P) \), respectively; \( M_i^P(A) \) denotes the set of action atoms with weight and level \( i \) that appear in \( A \), and \( \text{weight}(a) \) denotes the weight of resp. action atom \( a \).

Appendix B

An example of observe-think-act cycle with \textsc{acthex} programs

Using knowledge base update actions, \textsc{acthex} programs provide a convenient medium for modelling iteration and optimization tasks\(^{10}\). We can model optimization problems as a series of decision problems where each decision problem corresponds to an \textsc{ActHex} program. The action atoms can coordinate these series of decision problems to find the optimal solution. An example can be the maximal clique problem.

Let \( G \) be a graph encoded by means of \textit{vertex} and \textit{edge} predicates. A fact \textit{size}(\( n \)) (initially \( n = 1 \)) is put within a knowledge base called \textit{clique}, together with the following program:

\[
\begin{align*}
 &\textit{in}(X) \lor \neg \textit{in}(X) \leftarrow \textit{vertex}(X). \\
 &\leftarrow \textit{in}(X), \textit{in}(Y), \neg \textit{edge}(X,Y), X \neq Y. \\
 &\leftarrow \&\text{count}[\text{in}](X), X < N, \textit{size}(N). \\
 &\#\text{retract}[\textit{clique}, \textit{"size"}(N, 0)] \{b, 1\} \leftarrow \textit{size}(N). \\
 &\#\text{assert}[\textit{clique}, \textit{"size"}(M, 0)] \{b, 2\} \leftarrow \textit{size}(N), M = N + 1. \\
 &\#\text{execute}[\textit{clique}] \{b, 3\}. 
\end{align*}
\]

At each call to this program, we find whether there exist a clique of size \( N^{11} \). If a clique of such a size exists, the selected execution plan removes former assertions for the \textit{size} predicate and then the assertion \textit{size}(\( N + 1 \)) is pushed to \textit{clique} by means of an appropriate \#\text{assert} action: \textit{clique} is then executed again. The precedence value of the \#\text{execute} action atom is the largest among the other action atoms which ensures that reexecution of the program is issued after all the changes to the program are done. Notice that execution terminates for some size \( N' \) (which is the optimal clique size augmented by 1) for which \textit{clique} turns in an inconsistent program (having thus no execution schedules).

\(^{10}\) The example is illustrative of knowledge update action constructs only and is not to be considered as an alternative proposal for choices and other optimization construct known in ASP.

\(^{11}\) The external atom \&\text{count}[\text{in}](X) can be seen as as equivalent to the DLV (Leone et al. 2006) aggregate construct \#\text{count}\{Y : \text{in}(Y)\} = X.
Appendix C

An example of update to a Web source after reasoning under ASP semantics

The ACTHEX program below (in dlvhex native syntax) illustrates how to import knowledge from known web sources and perform subsequent reasoning; we exploit a #createEvent action predicate for updating the web sources at hand. Many constructs allowed in the dlvhex system are conveniently exploited, e.g. namespace declarations, the &rdf import predicate, string and aggregate external functions: the reader can find useful documentation about dlvhex constructs on the system web site\textsuperscript{12}.

%authentication information
user("acthex").
password("secretpassword").

#namespace(rdf, "http://www.w3.org/1999/02/22-rdf-syntax-ns#")
#namespace(ical, "http://www.w3.org/2002/12/cal/ical#")
#namespace(gcal, "http://www.google.com/calendar/feeds/")

meetingDate("2010-02-02"). % The extra apostrophe is necessary
%calendar ids of each team

googleCalendar(team1, "gcal:02ngn7n8s87fi6oojbn06sre4g@group.calendar.google.com/private/full").
googleCalendar(team2, "gcal:3h4m35be5g8q35hrb17dfq4ubk@group.calendar.google.com/private/full").

%rdf sources of each teams calendar
calendar(team1, "cals:02ngn7n8s87fi6oojbn06sre4g@group.calendar.google.com/public/basic").
calendar(team2, "cals:3h4m35be5g8q35hrb17dfq4ubk@group.calendar.google.com/public/basic").

%Getting rdf triples from calendars
calendarTriples(P, X, Y, Z) :- calendar(P, Q), &rdf{Q}(X, Y, Z).
event(M, X) :- calendarTriples(M, X, "rdf:type", "ical:Vevent").
aboutEvents(M, X, Y, Z) :- event(M, X), calendarTriples(M, X, Y, Z).

% Legenda
% M = Person Name (team1, team2, team3 ... )
% X = Event ID
% S = Event Start Time in ICAL format
% F = Event Finish Time in ICAL format
% T = Event type (transparent, opaque).
% Opaque Events have higher priority than transparent ones.
eventDetails(M, X, S, F, T) :- calendarTriples(M, X, "ical:dtstart", S1),
calendarTriples(M, S1, "ical:dateTime", S),
calendarTriples(M, X, "ical:dtend", F1),
calendarTriples(M, F1, "ical:dateTime", F),
calendarTriples(M, X, "ical:transp", T).

\textsuperscript{12} \url{http://www.kr.tuwien.ac.at/research/systems/dlvhex/}. 12
% Given two ternary predicates test range and busy in format (EventCode,StartTime,EndTime)
% overlap(E1,E2) returns whether events overlap with each other

nonoverlap(X, Y, Z) :- testRange(X, Sx, Fx), busy(Y, Sy, Fy, Z), Fx <= Sy.
nonoverlap(X, Y, Z) :- busy(X, Sx, Fx, Z), testRange(Y, Sy, Fy), Fx < Sy.
nonoverlap(X, Y, Z) :- nonoverlap(Y, X, Z).

overlap(X, Y, Z) :- testRange(X, Sx, Fx), busy(Y, Sy, Fy, Z), X >< Y, not nonoverlap(X, Y, Z).
overlap(X, Y, Z) :- busy(X, Sx, Fx, Z), testRange(Y, Sy, Fy), X >< Y, not nonoverlap(X, Y, Z).

% Select a meeting hour nondeterministically
any(X) :- inrange(X).
:- any(X), any(Y), X <> Y.
one :- nOne.

one :- inrange(X), any(X).

%Subprogram for finding the slots in which a participant is busy
% Legenda: busy(PersonID,StartTime,EndTime,TypeOfMeeting)

busy(M, S, F, T) :- meetingDate(D), eventDetails(M, X, S, F, T), split[S, “T”, 0](D).
baby(M, S, F, T) :- meetingDate(D), eventDetails(M, X, S, F, T), split[F, “T”, 0](D).

succ(Last, F) :- meetingDate(D),
&concat[D, “’T19:00:00Z’”](Last, &concat[D, “’T20:00:002’”](F).
succ(S, F) :- inrange(S), inrange(F), S < F, not someinthemiddle(S, F).
someinthemiddle(S, F) :- inrange(S), inrange(F), inrange(M), S < M, M < F.

chosenSlot(all, S) :- any(S).
testRange(all, S, F) :- chosenSlot(all, S), succ(S, F).

:- overlap(all, Y, “‘OPAQUE’”).[1 : 1]

#createEvent[Team, Url, “ActHexMeeting”, Date, User, Password{b, 1}]
    :- password(Password), user(User), googleCalendar(Team, Url), chosenSlot(_, Date).