Reasoning about Actions with Abductive Logic Programming*

Luís Moniz Pereira and Renwei Li
{lmp,renwei}@di.fct.unl.pt

Centro de Inteligência Artificial
Universidade Nova de Lisboa
2825 Monte de Caparica
Portugal

Abstract. In order to construct a computer-based system which can reason and act intelligently in the real world, we have to develop a computational but provably correct methodology and its related software system which can reason about actions and changes in dynamic domain. For this purpose we propose to use abductive logic programming paradigm as the computational mechanism. Technically, we make use of a simple, but extensible if needed, action description language to describe the domain in question. Then we use a sound and complete translation algorithm to transform domain descriptions into abductive logic programs. And thus reasoning about actions is reduced to abductive queries against abductive logic programs. In this paper we will only address three important issues: knowledge assimilation, refinement of action theories, and concurrent actions. For the task of knowledge assimilation we will introduce a formal and computational methodology, called the possible causes approach, in contrast to Ginsberg's possible worlds approach and Winslett's possible models approach. For the refinement of a possibly incomplete action theory, we use tests on the domain, and then abductively refine the original domain description to a new one which is closer to the domain in reality. For concurrent actions, we introduce a new semantics by using three-valued fluents to resolve conflicts among atomic actions.

1 Introduction

The development of computer-based systems, or so-called intelligent agents, which are able to observe, reason and act intelligently is one of the most challenging efforts ever made by Artificial Intelligence workers. It is hence no surprise that knowledge representation and reasoning techniques have become a focol topic in this field. Due to their declarative semantics and efficient execution, logic programs have become a more and more important tool for knowledge representation and reasoning. In this paper we will show the usefulness of abductive logic programming in reasoning about actions.

* Published in: *Proc. of EPIA'97, Portuguese International Conference in Artificial Intelligence*, Coimbra, 1997

Basically, an intelligent agent consists of at least three components: sensor for observing, effector for acting, and a processor for reasoning. The task of an intelligent agent can roughly be represented by an algorithmic loop:

- Select a goal G among all its intentions;
- Generate a (partial) plan P for the selected goal G;
- Perform an action in the plan P;
- Revise its knowledge base appropriately.

A central issue in the above loop is how to automate reasoning about actions and changes. In this paper we will address three related problems: knowledge assimilation, refinement of action theories, and concurrent actions. The result of this paper is currently used in constructing situated agents [21]. Technically, we start with a simple, but extensible if needed, action description language to describe the domain in question. Then we use a sound and complete translation algorithm to transform domain descriptions into abductive logic programs. And thus reasoning about actions is reduced to abductive queries against abductive logic programs. For the task of knowledge assimilation we will introduce a formal and computational methodology, called the possible causes approach (PCA), in contrast to Ginsberg's possible worlds approach (PWA) and Winslett's possible models approach (PMA). For the refinement of action theories, we start with a possibly incomplete action theory, use tests on the domain, and then abductively refine the original domain description to a new one which is closer to the domain in reality. For concurrent actions, we introduce a new semantics by using three-valued fluents to resolve conflicts among atomic actions.

2 Domain Descriptions

When reasoning about actions and changes, we often assume that an action theory has been given and described in a formal language or in a framework, e.g. situation calculus [24], event calculus [19], action description languages $\mathcal{A}$ [14] and ADL [25], the fluent-features framework (FFF) [29], and their variants or extensions.

In this section we introduce an action description language $\mathcal{A}^+$, an extension to $\mathcal{A}$ of [14]. And thus $\mathcal{A}^+$ is one of $\mathcal{A}$ family languages. The reason we choose $\mathcal{A}$ is simply that $\mathcal{A}$ has been shown to be a simple, extensible and expressive action description language, and to be equivalent to other three major formalisms [18] proposed by Pednault [25], Reiter [28] and Baker [6], respectively.

$\mathcal{A}^+$ will be directly used to represent a possibly incomplete action theory to be refined. A complete action theory can be represented in $\mathcal{A}$. And hence $\mathcal{A}$ will be used for knowledge assimilation. We will also extend $\mathcal{A}$ with concurrent actions later.

2.1 Syntax

We begin with three disjoint non-empty sets of symbols, called *proposition names*, *fluent names*, and *action names*, respectively. For convenience we will also use parameterized names. *Actions* and *propositions* are defined to be action names and proposition names, respectively. A *fluent expression*, or simply *fluent*, is defined to be a fluent name possibly preceded by $\neg$. A fluent expression is also called a *positive fluent* if it only consists of a fluent name; otherwise it is called a *negative fluent*.

In $\mathcal{A}^{+}$, a domain description is defined to be a set of effect assertions and constraints. An effect assertion is defined to be a statement of the form

$$A \textbf{ causes } F \textbf{ if } P_1, \ldots, P_m, Q_1, \ldots, Q_n$$

where A is an action, each of F, P_1, ..., P_m $(m \geq 0)$ is a fluent expression, and each of $Q_1, \ldots, Q_m$ $(n \geq 0)$ is a proposition name. If $m = n = 0$, then we will simply write it as A **causes** F. A constraint is defined as follows:

- A proposition name is an atomic constraint.
- A statement of the form

  $$F \textbf{ after } A_1, \ldots, A_n$$

  where F is a fluent and A_i is an action, is an atomic constraint, also called *value assertion*. When $n = 0$, the value assertion above is abbreviated to **initially** F.
- If C_1 and C_2 are constraints, then $\neg C_1$, $C_1 \wedge C_2$, $C_1 \vee C_2$ are constraints, called complex constraints. Other propositional connectives can be defined in terms of them as derived connectives.

It can be seen that $\mathcal{A}^{+}$ is an extension of $\mathcal{A}$ by allowing propositions and more types of constraints.

2.2 Remarks

It seems that we would increase the expressive power if we defined the effect assertions in the following way: (1) A basic effect assertion is a statement of the form A **causes** F **if** $C_1, \ldots, C_n$; (2) An effect assertion is a statement of the form $(E_{11} \wedge \ldots \wedge E_{1m_1}) \vee \ldots \vee (E_{n1} \wedge \ldots \wedge E_{nm_n})$, where each E_{ij} is a basic effect assertion. In fact, combining with proposition names, we can reduce the above complex effect assertion to simpler ones of $\mathcal{A}^{+}$. We can systematically do so by introducing a few new proposition names and then transform effect assertions. For example, consider:

$$\begin{array}{l} (A_1 \textbf{ causes } F_1 \textbf{ if } C_{11}, \ldots, C_{1n_1}) \\ \vee \ldots \\ \vee (A_m \textbf{ causes } F_m \textbf{ if } C_{m1}, \ldots, C_{mn}) \end{array}$$

Let h_i, $1 \leq i \leq m$ be m new proposition symbols. Then, the above complex effect assertions can be transformed into m basic effect assertions and a new constraint as follows:

$$\begin{array}{l} A_1 \textbf{ causes } F_1 \textbf{ if } C_{11}, \ldots, C_{1n_1}, h_1 \\ \ldots \\ A_m \textbf{ causes } F_m \textbf{ if } C_{m1}, \ldots, C_{mn}, h_m \\ h_1 \vee \ldots \vee h_m \end{array}$$

On the other hand, it also seems that we would increase the expressive power if we allowed general well-formed propositional formulas in the preconditions of effect assertions. For example, let A be an action, P_1 a fluent, and Q_1, Q_2, Q_3 be proposition names. Consider

$$A \textbf{ causes } F \textbf{ if } P_1, (Q_1 \wedge Q_2) \vee \neg Q_3$$

This kind of seemingly more expressive effect assertions can also be reduced to effect assertions in $\mathcal{A}^+$. Let Q_4 be a new proposition name. The following effect assertion and a constraint is equivalent to the above assertion:

$$\begin{array}{l} A \textbf{ causes } F \textbf{ if } P_1, Q_4 \\ Q_4 \leftrightarrow (Q_1 \wedge Q_2) \vee \neg Q_3 \end{array}$$

2.3 Semantics

The semantics of a domain description is defined by using proposition assignment, states, and transitions.

A proposition assignment α is a set of proposition names. Given a proposition name P and an assignment α, we say that P is true if $P \in \alpha$, and $\neg P$ is true if $P \notin \alpha$. A *state* is a set of fluent names. Given a fluent name F and a state σ, we say that F holds in σ if $F \in \sigma$; $\neg F$ holds in σ if $F \notin \sigma$. A *transition function* Φ is a mapping from the set of pairs (A, σ), where A is an action expression and σ is a state, to the set of states.

An *interpretation structure* is a triple (α, σ_0, Φ), where α is an assignment, σ_0 is a state, called the *initial state* of (σ_0, Φ), and Φ is a transition function. For any interpretation structure $M = (\alpha, \sigma_0, \Phi)$ and any sequence of action expressions $A_1; \ldots; A_m$ in M, by $\Phi(A_1; \ldots; A_m, \sigma_0)$ we denote the state $\Phi(A_m, \Phi(A_{m-1}, \ldots, \Phi(A_1, \sigma_0) \ldots))$.

Given an interpretation structure (α, σ_0, Φ), a constraint C is said to be true with respect to it iff

- if C is a proposition name, then $C \in \alpha$;
- if C is a value assertion of the form F **after** $A_1, \ldots, A_n$, then F holds in the state $\Phi(A_1; \ldots; A_n, \sigma_0)$;
- if C is a complex constraint, then it is true according to the usual propositional connective evaluation methods.

An interpretation structure (α, σ_0, Φ) is a *model* of a domain description D iff

- Every constraint is true with respect to the interpretation structure.

– For every action A, every fluent name F, and every state σ: (i) If D includes an effect assertion A **causes** F **if** $P_1, \ldots, P_m, Q_1, \ldots, Q_n$, such that fluents P_1, ..., P_m hold in σ and propositions $Q_1, \ldots, Q_n$ are true with respect to (α, σ_0, Φ), then $F \in \Phi(A, \sigma)$; (ii) If D includes an effect assertion A **causes** $\neg F$ **if** $P_1, \ldots, P_m, Q_1, \ldots, Q_n$, such that fluents P_1, ..., P_n hold in σ and propositions $Q_1, \ldots, Q_n$ are true with respect to (α, σ_0, Φ), then $F \notin \Phi(A, \sigma)$; (iii) If D does not include any such effect assertions, then $F \in \Phi(A, \sigma)$ iff $F \in \sigma$.

A domain description is *consistent* if it has a model. A domain description is *complete* if it has exactly one model. A domain description D entails a value assertion V if V is true in all models of D. It can be shown that different models of the same domain description differ only in different initial states and/or proposition assignments. In addition, the interpretation of a proposition name is independent of states.

In reality a practical domain should have only one model. The task of refining domain descriptions is to construct a new domain description which has fewer models than the original domain description. We will achieve this purpose by first performing some actions and observing their outcome, then we will abductively infer the truth values of propositions and initial states. We will make use of abductive logic programming for the purpose of abductive reasoning.

3 Abductive Logic Programming

As will be seen, the translations of domain descriptions will be acyclic, and thus In this paper we will follow Denecker [12] and use Console's predicate completion semantics by only completing defined predicates and leaving abducible predicates open.

An abductive logic program is a triple $< P, IC, \Delta >$, where P is a set of logic programming rules, IC is a set of first-order sentences as integrity constraints, and Δ is a set of predicates, called abducible predicates. A logic programming rule is of the form

$$A \leftarrow B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_n$$

where A and each B_i are atoms, and *not* is the negation-as-failure operator.

An abductive answer δ to a query Q in $< P, IC, \Delta >$ is a finite subset of instances of Δ such that (i) $Q \in SEM(P \cup \{a \leftarrow\ : a \in \delta\}, IC)$; (ii) $P \cup \{a \leftarrow\ : a \in \delta\}, IC)$ is consistent according to definition of SEM; (iii) δ is minimal in the sense that no subset of it satisfies the previous two conditions, where $SEM(P, IC)$ denotes the semantics of the program P with constraints IC.

There have been a few competing semantics for logic programs in the literature: predicate completion semantics, stable model semantics [13], and well-founded model semantics [31]. All of them can be extended for abductive logic programs. As is known, for acyclic programs [2] [5], their predicate completion models coincide with both the stable models and the well-founded models. In the case of abductive logic programs, the semantical coincidence still holds, as shown by Denecker [10]. In this paper we will only use acyclic logic programs, and will define the semantics of logic programs as the predicate completion semantics. For abductive logic programs, we will complete all predicates except the abducible ones [8].

Example 3.1 *Consider an abductive logic programming framework* $FOO = (P, A, I)$*, where*

$$P = \left\{ \begin{array}{lcl} p & \leftarrow & a \\ p & \leftarrow & b \\ q & \leftarrow & b \end{array} \right\}, \quad A = \{a, b\}, \quad I = \{false \leftarrow p, q\}$$

Then, we have

$$COMP(P; A) = \left\{ \begin{array}{l} p \leftrightarrow a \vee b \\ q \leftrightarrow b \end{array} \right\}$$

The semantics of FOO *is* $COMP(P; A) \cup I$*, which is equivalent to the following first-order theory*

$$\{p \leftrightarrow a \vee b, q \leftrightarrow b, \neg(p \wedge q)\}$$

Suppose that we want to evaluate a query $? - p$*. Then* $\{a\}$ *is an abductive answer, since it satisfies the following three conditions:*

- $COMP(P; A) \cup I \cup \{a\} \models p$;
- $COMP(P; A) \cup I \cup \{a\}$ *is consistent.*
- *There is no proper subset of* $\{a\}$ *satisfying the above two conditions,and hence* $\{a\}$ *is a set-inclusion minimal subset satisfying the above two conditions.*

Note that $COMP(P; A) \cup I \cup \{b\}$ *is not consistent. Thus,* $\{b\}$ *is not an abductive answer to* $? - p$ *in* FOO*. Now suppose that we want to evaluate a query* $? - q$*. It can be shown that there is no abductive answer to it.*

There have been many proposals, e.g., [11, 30], for abductive query evaluation procedures for abductive normal logic programs. The work reported in this paper has been experimented with the latest version of the REVISE system [9], an extended logic programming system for revising knowledge bases. It is based on

[2] Let P be a normal logic program, and B_P the set of ground atoms of P. A *level mapping* for a program P is a function $\lambda : B_P \rightarrow N$ from ground atoms to natural numbers. For $A \in B_P$, $\lambda(A)$ is the *level* of A; Given a level mapping λ, we extend it to ground negative literals by letting $\lambda(\neg A) = \lambda(A)$; A clause of P is called *acyclic with respect to* a level mapping λ, if for every ground instance $A \leftarrow L_1, \ldots, L_n$ of it, we have $\lambda(A) > \lambda(L_i)$, for every $1 \leq i \leq n$. A program P is called em acyclic with respect to a level mapping λ, if all its clauses are. P is called *acyclic* if it is acyclic with respect to some level mapping.

logic programming with explicit negation and integrity constraints and provides a two- or three-valued assumption revision to remove contradictions from the knowledge base. The latest version of the REVISE system is based on a top-down query evaluation procedure [2] of well founded semantics with explicit negation (WFSX) [26, 4]. If explicit negation is not used, then the WFSX semantics coincides with the WFS semantics. Thus, the use of REVISE is not essential for the purpose of this paper.

4 Refinement of Action Theories

4.1 Motivation

When reasoning about actions and changes, we often assume that a complete action theory has been given and described in a formal language. But sometimes no complete action theory is available in practice. We may need to do some tests and perform some reasoning in order to have a complete action theory. For example, let's consider Vladimir Lifschitz' challenge problem[3]:

> The room has two lamps, say Big and Small, and two light switches, say Left and Right. A switch controls one and only one light. Both lights are off. Initially we don't know whether the wiring is this way or the other way around, but we can find out by toggling a switch.

In this example, we have two actions: to toggle the left switch and to toggle the right switch, denoted by $toggle(left)$ and $toggle(right)$, and we have two fluents: the big light is on and the small light is on, denoted by $on(big)$ and $on(small)$. If we knew the way in which the circuit is connected, then we could generate plans, predict the future, or explain the past. The problem is that no such an immediately available theory is available. An intelligent agent should be able to perform some tests and then obtain a complete action theory. In this paper we will present an abductive methodology for reasoning about actions and changes starting from an incomplete action theory, i.e., an action theory with more than one model, then refining it by testing and abductive reasoning so as to have a complete action theory, which can then be used for planning, predicting and explaining depending on specific needs. Our methodology consists of a high-level action description language $\mathcal{A}^+$, a translation from $\mathcal{A}^+$ to abductive logic programs, and an abductive logic programming system used as the underlying inference engine for refinement. We will return to this problem later.

4.2 Translation into Abductive Programs

In this section we will present a translation from domain descriptions in $\mathcal{A}^+$ into abductive logic programs.

Let D be a domain description. The translation πD includes a set of programming rules and a set of constraints defined as follows:

[3] Vladimir Lifschitz's email message to lmp@di.fct.unl.pt and renwei@di.fct.unl.pt on March 25, 1996.

1. Initialization: $holds(F, s_0) \leftarrow initially(F)$.
2. Law of Inertia:

$$holds(F, result(A, S)) \leftarrow holds(F, S), not\ noninertial(F, S, A).$$

   where *not* is the negation-as-failure operator. By the law of inertia, F is true at a new situation by doing A on S if it was true at S.
3. Each effect assertion a **causes** f **if** $p_1, \ldots, p_m$, $q_1, \ldots, q_n$, with f being positive, p_i being a fluent, and q_i being a proposition, is translated into

$$holds(f, result(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_m, S), q_1, \ldots, q_n.$$

   where $holds(\neg p, S)$ with p being positive stands for *not* $holds(p, S)$. This convention is also used in the rest of this paper.
4. Each effect assertion a **causes** $\neg f$ **if** $p_1, \ldots, p_m$, $q_1, \ldots, q_n$, with f being positive, p_i being a fluent, and q_i being a proposition, is translated into

$$noninertial(f, a, S) \leftarrow holds(p_1, T), \ldots, holds(p_m, T), q_1, \ldots, q_n.$$

5. For every constraint C of D: (i) if C is a proposition name, $\pi C = C$; (ii) if C is f **after** $a_1, \ldots, a_n$ with f being positive, then $\pi C = holds(f, result(a_1; \ldots; a_n, s_0))$; (iii) if C is $\neg f$ **after** $a_1, \ldots, a_n$, with f being positive, then $\pi C = \neg holds(f, result(a_1; \ldots; a_n, s_0))$; (iv) $\pi(\neg C_1) = \neg(\pi C_1)$, $\pi(C_1 \wedge C_2) = \pi C_1 \wedge \pi C_2$, $\pi(C_1 \vee C_2) = \pi C_1 \vee \pi C_2$.

We will define abducible predicates to be $initially(F)$ and all proposition names. The semantics of πD, denoted by $Comp(\pi D)$, is defined to be the first-order theory by completing all predicates except $initially(F)$ and proposition names, jointly with Clark's theory of equality, and the constraints [8].

Theorem 4.1 *Let D be any domain description in $\mathcal{A}^+$. πD is an acyclic logic program with first-order constraints in the sense of [5].*

Corollary 4.2 *The completion semantics $Comp(\pi D)$ of πD agrees with its generalized stable model semantics [17] and generalized well-founded model semantics [27].*

The above corollary means that the result of this paper can be experimented with any abductive logic programming system with one of the three major semantics. The detailed proof follows from [10].

4.3 Soundness and Completeness

In general it is very difficult to reason about actions in $\mathcal{A}^+$. The purpose of the translation is to reduce the reasoning work in $\mathcal{A}^+$ to abductive querying in an abductive logic programming system. This section will show that reasoning in $\mathcal{A}^+$ is equivalent to abductive querying.

Theorem 4.3 *The translation π is sound. That is, for any domain description D and any value assertion V, if $Comp(\pi D) \models \pi V$, then D entails V.*

Definition 4.4 *A domain description D is effect consistent iff for each pair of effect assertions,*

$$\begin{array}{l} A \ \textbf{causes} \ \ F \ \textbf{if} \ \ C_1, \ldots, C_m \\ A \ \textbf{causes} \ \ \neg F \ \textbf{if} \ \ C_{m+1}, \ldots, C_n \end{array}$$

in D, there exists i, $1 \leq i \leq m$, and j, $m+1 \leq j \leq n$, such that C_i is the complement of C_j.

Theorem 4.5 *The translation π is complete for any effect consistent domain descriptions. That is, for any effect consistent domain description D and any value assertion V, if D entails V, then $Comp(\pi D) \models \pi V$.*

The requirement for a domain description to be effect consistent is necessary. If a domain description D is not effect consistent, no transition functions exist to satisfy its effect assertions, thus it has no models, and hence it entails every value assertion. On the other hand, its translation is consistent and thus has at least one model which entails a proper subset of what D entails.

The above soundness and completeness theorems signify that our translation can actually be used for the general purposes of reasoning about actions and changes such as abductive planning, prediction, explanation. That is to say, our result of this paper goes beyond refinement of action theories. But we will not delve into detailed discussion on how to use our translation for abductive planning, temporal prediction and explanation. In the next section we will concentrate on refinement of action theories.

4.4 Refinement

Let D be a domain description. D may have more than one model. If D has more than one model, we may only predict a disjunctive future instead of a definite future. That is to say, after a sequence of actions is done, we cannot predict whether a fluent is definitely true or not. When a domain description is complete, we can always predict whether a fluent is true or not after an action is done. This is sometimes a very important factor in reasoning about actions, as shown as in [22].

When a domain description is not complete, all its models differ in their initial states and/or proposition assignments. In order to determine initial states and proposition assignments, one may perform some tests: doing some actions, observing their effects, and then abductively determining initial states and proposition names.

Definition 4.6 *Let D_1 and D_2 be two domain descriptions. D_2 is said to be a refinement of D_1 iff every model of D_2 is a model of D_1.*

Now suppose that we are given a domain description D_0. We want to refine it. The way to do it, as said as before, is to perform some actions and observe their effects. This process is called *test*. The purpose of tests is to generate new value assertions. And thus we can formally define a test to be a set of value assertions.

Definition 4.7 *A* test *in an action domain is a set of value assertions. Let D be a domain description. The pair (D, τ) is called a refinement problem.*

Theorem 4.8 *Let D be a domain description, and τ a test. Then, $D \cup \tau$ is a refinement of D.*

The above theorem means that simply adding tests to a domain description will definitely refine a domain description. But syntactically $D \cup \tau$ is more complicated than D. We may prefer simpler and finer descriptions. Note that in an interpretation structure, all proposition names will be either true or false. In the reality, all these proposition names can and can only be either true or false. When we do enough tests, the refinement of the domain will be closer and closer to a complete domain description. The above corollary implies that the complete domain description is a limit of all refinements of domain descriptions. When the domain description has only one model, all proposition names can be removed from the domain description by substituting them with their truth values, and thus syntactically simplifying the domain description. Hence, the previous definition of refinements is modified as follows.

Definition 4.9 *Let D_1 and D_2 be two domain descriptions. D_2 is said to be a refinement of D_1 iff the following conditions are satisfied:*

- *Every model of D_2 is a model of D_1;*
- *There is no proposition name in D_2 which is true in every model of D_2;*
- *There is no proposition name in D_2 which is false in every model of D_2.*

In what follows we want to show how to compute refinements with abductive logic programming. In Section 4.2 we presented a translation from domain descriptions to abductive logic programs. However, many existing abductive logic programming systems do not directly support our constraints. Instead, they support constraints of the form

$$\perp \leftarrow L_1, \ldots, L_n$$

First we need to translate all constraints into the above form.

The translation, still denoted by π, is as follows. Let C be a constraint in the program πD. Then C can be equivalently transformed into a conjunctive normal form:

$$(C_{11} \vee \ldots \vee C_{1m_1}) \wedge \ldots \wedge (C_{m1} \vee \ldots \vee C_{mn})$$

Then, it will be translated into

$$\bot \leftarrow \textit{not } C_{11}, \ldots, \textit{not } C_{1m_1}$$
$$\ldots$$
$$\bot \leftarrow \textit{not } C_{m1}, \ldots, \textit{not } C_{mn}$$

where *not* $\neg L$ is taken as L.

After constraints are translated into a logic program, we can run it in any abductive logic programming system. Before proceeding, we need to guarantee that the correctness of the translation is preserved.

Theorem 4.10 *The translation π is both sound and complete for any effect consistent domain descriptions.*

Note that above translation does not change completions of non-abducible predicates. On the other hand, the completion of $\bot$ is logically equivalent to $\neg C$. Therefore, the semantics of new programs is the same as before.

Let $\tau = \{V_1, \ldots, V_n\}$ be a test. Then, τ can be transformed into a query:

$$\leftarrow \pi V_1, \ldots, \pi V_n$$

where for each i, πV_i is defined as follows: Let V_i be F **after** $A_1, \ldots, A_n$ in τ. If F is positive, then πV_i is defined to be $holds(F, result(A_1; \ldots; A_n, s_0))$; if F is negative and equal to $\neg G$, then πV_i is defined to be *not* $holds(G, result(A_1; \ldots; A_n, s_0))$.

Submitting the query to an abductive logic programming system, we will get abductive answers to it. In what follows we will write $\mathcal{R}(D, \tau)$ to stand for the set of all abductive answers to the query $\leftarrow \pi\tau$ against the abductive logic program πD. Now we are in a position to define the procedure of refining action theories.

Definition 4.11 *Let D be a domain description and τ a test. Let $\mathcal{R}(D, \tau) = \{R_1, \ldots, R_n\}$. Perform:*

1. *For every proposition name P, if $P \notin R_1 \cup \ldots \cup R_n$, remove from D all effect assertions containing P in the precondition list, and replace P with $false$ in every constraint of D;*
2. *For every proposition name P, if $P \in R_1 \cap \ldots \cap R_n$, remove P from all effect assertions of D, and replace P with $true$ in every constraint of D;*
3. *Simplify constraints of D in the usual way by using of $true$ and $false$ in the formulas. For example, if C is of the form $\neg false$ or $C_1 \vee true$, C is removed.*

Then, define $\mathcal{S}(D, \tau)$ to be the set of the resulting effect assertions, constraints, and the test τ.

The following theorem says that the new domain description $\mathcal{S}(D, \tau)$ is a refinement of D.

Theorem 4.12 *Let D be a domain description, τ a test. Then, $\mathcal{S}(D, \tau)$ is a refinement of D.*

4.5 An Example

Now we return to the example in the Introduction. Let $controls(S, L)$ be a parameterized proposition name to denote that switch S controls light L. Then, we can have the following domain description D:

$controls(left, small) \leftrightarrow controls(right, big)$
$controls(left, big) \leftrightarrow controls(right, small)$
$controls(left, small) \dot{\vee} controls(left, big)$
$controls(right, small) \dot{\vee} controls(right, big)$
$toggle(left)$ **causes** $on(small)$ **if** $\neg on(small), controls(left, small)$
$toggle(left)$ **causes** $\neg on(small)$ **if** $on(small), controls(left, small)$
$toggle(right)$ **causes** $on(small)$ **if** $\neg on(small), controls(right, small)$
$toggle(right)$ **causes** $on(small)$ **if** $\neg on(small), controls(right, small)$
$toggle(left)$ **causes** $on(big)$ **if** $\neg on(big), controls(left, big)$
$toggle(left)$ **causes** $\neg on(big)$ **if** $on(big), controls(left, big)$
$toggle(right)$ **causes** $on(big)$ **if** $\neg on(big), controls(right, big)$
$toggle(right)$ **causes** $on(big)$ **if** $\neg on(big), controls(right, big)$
initially $\neg on(big)$
initially $\neg on(small)$

Then, we have an abductive logic program πD. Now suppose we have a test τ $= \{on(big)$ **after** $toggle(left)\}$. Then, evaluate the following query against the abductive logic program πD:

$$\leftarrow holds(on(big), result(toggle(left), s_0))$$

An abductive logic programming system should correctly output the following unique abductive answer $\mathcal{R}(D, \tau)$:

$$\{\{controls(right, small), controls(left, big)\}\}$$

Then, by definition we have the following new domain description $\mathcal{S}(D, \tau)$:

$toggle(right)$ **causes** $on(small)$ **if** $\neg on(small)$
$toggle(right)$ **causes** $on(small)$ **if** $\neg on(small)$
$toggle(left)$ **causes** $on(big)$ **if** $\neg on(big)$
$toggle(left)$ **causes** $\neg on(big)$ **if** $on(big)$
initially $\neg on(big)$
initially $\neg on(small)$
$on(big)$ **after** $toggle(left)$

Thus we have obtained a complete domain description which enables us to generate plan, to predict the future, or to explain the past, as what we expected and intended.

5 Knowledge Assimilation

5.1 Motivation

Knowledge assimilation is a process of reasonably incorporating new knowledge into an existing knowledge base. It is not always clear how different knowledge assimilation and belief revision/update. Existing approaches to belief revision/update can be roughly classified into postulate-based approaches such as AGM theory [1], syntax-based approaches such as the possible worlds approach PWA [16], and semantics-based approaches such as the possible models approach PMA [32]. The AGM theory is rooted in philosophical logic, and the syntax/semantics-based approaches seem to be currently favored in the AI community since they can be immediately used to solve some practive problems such as reasoning about actions, diagnosis, counterfactual query, etc.

Now consider a domain where there are two people: Tom and $Mary$, and two objects: $Money$ and $Book$. In the beginning, say November 1, 1995, Tom has $Book$ while $Mary$ has $Money$. Assume that either of the two people can buy $Book$ from the other if he/she has $Money$ and the other has $Book$; either can give $Book$ or $Money$ to the other; either can steal an object from the other. Suppose that $Mary$ is found to have $Book$ later, say November 5, 1995.

Applying either PWA or PMA, we will only have the following knowledge base [20]:

$$\{\neg has(Tom, Book), \neg has(Tom, Money), has(Mary, Book), has(Mary, Money)\}$$

However, in practice there is another possible new knowledge base:

$$\{\neg has(Tom, Book), has(Tom, Money), has(Mary, Book), \neg has(Mary, Money)\}$$

Intuitively, the former corresponds to the possible cause: Tom gives $Book$ to $Mary$ or $Mary$ steals $Book$ from Tom, while the latter corresponds to the possible cause: $Mary$ buys $Book$ from Tom with Mary's $Money$. These two possible causes should be equally possible. In our PCA approach, both of the new knowledge bases can be found.

We will make use of the following sub-languge of $\mathcal{A}^+$ to describe a domain. The effects of actions are described by effect assertions of the form

$$A \textbf{ causes } F \textbf{ if } P_1, \ldots, P_n$$

The initial conditions are described by the value assertions of the form

$$\textbf{initially } F$$

A domain description is a set of the above two types of assertions.

Note that such a domain description can be obtained through refinement of action theories. In the following we first translate such a domain description into abductive logic programs including a time dimension.

5.2 Translation into logic programs

In the following we present a new translation π from domain descriptions to abductive logic programs, where a time dimension is incorporated so that one can represent and reason about narratives, actions and time. Let D be a domain description, the translation πD consists of the following logic programming rules:

1. Time dimension:
   In this paper we assume that time is structured on points, represented by a totally linearly-ordered set $(TP, \prec, succ, init)$, where TP is an infinite set of time points, $init \in TP$, and $succ(T) \in TP$ for any $T \in TP$. We will often use natural numbers which have a straightforward correspondence to terms for time points: $init$ corresponding to 0, $succ(init)$ to 1, etc.
2. Initialization: $holds(F, init) \leftarrow initially(F)$.
3. Law of Inertia:
$$\begin{aligned} holds(F, succ(T)) \leftarrow\ & holds(F, T), \\ & not\ noninertial(F, T). \end{aligned}$$
   where not is the negation-as-failure operator. By the law of inertia, a fluent F is true at a time point T if it was true earlier and inertial then.
4. Each e-proposition a **causes** f **if** $p_1, \ldots, p_n$, with f being positive, is translated into
$$\begin{aligned} holds(f, succ(T)) \leftarrow\ & happens(a, T), \\ & holds(p_1, T), \ldots, holds(p_n, T). \end{aligned}$$
5. Each e-proposition a **causes** $\neg f$ **if** $p_1, \ldots, p_n$,
   with f being positive, is translated into
$$\begin{aligned} noninertial(f, T) \leftarrow\ & happens(a, T), \\ & holds(p_1, T), \ldots, holds(p_n, T). \end{aligned}$$
6. Each v-proposition **initially** f or **initially** $\neg g$, where f and g are positive, is respectively translated into
$$\begin{aligned} false &\leftarrow not\ holds(f, init). \\ false &\leftarrow holds(g, init). \end{aligned}$$
   The above two logic programming rules function as integrity constraints.

5.3 Representation of Knowledge Bases

In this section, we we will use the above translation to represent temporal knowledge bases about domains of actions.

5.4 Temporal knowledge bases

Given a domain description D, we want to determine whether a fluent holds at a particular time point after some actions have happened in the past. We start with the concept of histories, then employ abductive logic programming for the representation of temporal knowledge.

A domain history H is a finite set of pairs (A, T) for action A and time T. By $(A, T) \in H$ we mean that A happens at T. In this paper we do not consider concurrent actions. Thus we assume that for any $T \in TP$ there is at most one action A such that $(A, T) \in H$.

A domain evolution E is a pair (D, H), where D is a domain description and H a history. Given a domain evolution (D, H), we can have an abductive logic program $KB(D, H)$ defined as follows:

$$KB(D, H) = \pi D \cup \{false \leftarrow not\ happens(a, t) : (a, t) \in H\}$$

The logic program $KB(D, H)$ will be simply called *the knowledge base generated from D and H*. The literal $false$ is always interpreted as logical falsity, and all rules with $false$ as heads function as integrity constraints. The predicates $initially(F)$ is taken as an abducible predicate used to capture the incomplete knowledge about the initial situation/time. The semantics of $KB(D, H)$ is defined to be the union of the integrity constraints, the Clark Equality Theory, and the first-order theory obtained by completing all the non-abducible predicates (all predicates except $initially(F)$). We will simply write $COMP(KB(D, H))$ to denote the semantics of of $KB(D, H)$, and often write $KB(D, H) \models Q$ to stand for $COMP(KB(D, H)) \models Q$. The following two results justify the above semantics definition.

Proposition 5.1 *$KB(D, H)$ is an acyclic logic program.*

Corollary 5.2 *The semantics $COMP(KB(D, H))$ of the abductive logic program $KB(D, H)$ coincides with its generalized stable model semantics and generalized well-founded model semantics.*

5.5 Soundness and completeness

Given a history H, we can have a unique sequence of actions $A_1; A_2; \ldots; A_m$ and a unique sequence of time points $T_1 \prec T_2 \prec \ldots \prec T_m$ such that $H = \{(A_i, T_i) : 1 \le i \le m\}$. The sequence of actions $A_1; A_2; \ldots; A_m$ will be called *the trajectory generated from H, denoted by $Traj(H)$*. A pre-history of H is defined to be a subset H' of H such that $Traj(H')$ is a prefix of $Traj(H)$.

Given an interpretation structure $M = (\sigma_0, \Phi)$, a history H and a time point T, we can derive a state S as follows: if $H' = \{(A_1, T_1), \ldots, (A_k, T_k)\}$ is a pre-history of H such that $T_k \prec T$ and there is no time point Tx and action Ax with $T_k \prec Tx \prec T$ and $(Ax, Tx) \in H$, then $S = \Phi(A_1; A_2; \ldots; A_k, \sigma_0)$, called *the state generated from H and T in M*, denoted by $GS_M(H, T)$. If T is a time point such that $(A, T) \notin H$ for any action A, and $(A, Tx) \notin H$ for any action A and any time point Tx with $T \prec Tx$, then the state $GS_M(H, T)$ is called *the state* *generated from H in M*, denoted by $GS_M(H)$. For example, let $M = (\sigma_0, \Phi)$ be an interpretation structure, and $H = \{(a,1),(a,6),(b,4),(c,10)\}$ a history. Then, $GS_M(H,5) = \Phi(a;b,\sigma_0)$ and $GS_M(H) = \Phi(a;b;a;c,\sigma_0)$.

Proposition 5.3 *Given an interpretation structure M, a history H, and a time point T, the state $GS_M(H,T)$ generated from H and T in M is unique.*

Note that $\mathcal{A}$has an infinite tree-like time structure. Given a history H, the translation π only takes into account a special path in the tree corresponding to H. In order for π to be correct, we require that π be correct with respect to any history.

Definition 5.4 *Let D be a domain description, and H a history. The logic program $KB(D,H)$ is said to be sound with respect to H iff, for every model M of D, fluent F and time T, if $KB(D,H) \models holds(F,T)$, then $F \in GS_M(H,T)$. The translation π is said to be sound iff, for every history H, $KB(D,H)$ is sound with respect to H.*

Definition 5.5 *Let D be a domain description, and H a history. The logic program $KB(D,H)$ is said to be complete with respect to H iff, for every fluent F and time T, if $F \in GS_M(H,T)$ for every model M of D, then $KB(D,H) \models holds(F,T)$. The translation π is said to be complete iff, for every history H, $KB(D,H)$ is complete with respect to H.*

Theorem 5.6 (Soundness) *Assume that D is a domain description. For any history H, model I of D, fluent F, and time T, if $KB(D,H) \models holds(F,T)$, then $F \in GS_I(H,T)$.*

A domain description D is *e-consistent* [12] iff for each pair of e-propositions a **causes** f **if** $p_1,\ldots,p_n$ and a **causes** $\neg f$ **if** $p_{n+1},\ldots,p_m$ in D, there exist i and j such that p_i is the complement of p_j.

Theorem 5.7 (Completeness) *Assume that D is an e-consistent domain description. For any history H, fluent F, and time T, if $F \in GS_I(H,T)$ for any model I of D, then $KB(D,H) \models holds(F,T)$.*

5.6 Possible Causes Approach

For easy reference, we will simply write $KB(D,H) \oplus N$ to informally denote all the possible new knowledge bases obtained by incorporating N into $KB(D,H)$ before we formally define it later. For convenience, we often write $N = L$ if $N = \{L\}$.

Suppose that $N = holds(F,T)$ is a newly observed fact. In the process of incorporating N into $KB(D,H)$, first the agent should check whether N is already stored in her knowledge base $KB(D,H)$, i.e., whether $KB(D,H) \models N$. If it is so, the agent will do nothing. In this case, $KB(D,H) \oplus N \equiv KB(D,H)$. Suppose that N is not stored in the knowledge base, that is, $KB(D,H) \not\models N$.

In this case, either the fluent F was true and inertial at the last time point or an action has happened at the last time point and a initiates F. After a careful analysis, it can be seen that the v-propositions **initially** F and occurrences of actions which initiate N or recursively initiate preconditions of actions which initiate N are possible causes for N in $KB(D,H)$.

Let (D,H) be a domain evolution, and Δ the set of instances of $initially(F)$ and $happens(A,T)$. For any finite subset δ of Δ we will write $KB(D,H) \dot{+} \delta$ to stand for $KB(D',H')$, where $D' = D \cup \{$ **initially** $F : initially(F) \in \delta\}$ and $H' = H \cup \{(A,T) : happens(A,T) \in \delta\}$. We assume that no concurrent actions appear in H'. We will often omit $initially(F)$ from δ when both **initially** $F \in D$ and $initially(F) \in \delta$.

Definition 5.8 *Let D be a domain description, H a history, and N a set of literals. A possible cause for N in $KB(D,H)$ is a subset δ of Δ such that (i) $KB(D,H) \dot{+} \delta \models N$; (ii) $KB(D,H) \dot{+} \delta$ is consistent; (iii) For any time T, there are no different actions A and B such that $(A,T),(B,T) \in H \cup \{(C,T) : happens(C,T) \in \delta\}$; (iv) δ is minimal in the sense that no subset δ' of δ exists such that δ' satisfies the first three conditions. We will write $KB(D,H) \uparrow N$ to stand for all the possible causes for N in $KB(D,H)$.*

As an example, consider the Stolen Car Problem:

$$D_{scp} = \{ \textbf{ initially } \neg Stolen.\ Steal \textbf{ causes } Stolen.\}$$

and $H_{scp} = \emptyset$. Suppose that the car is missing from the parking lot at time 3. Then there are three possible causes for $holds(stolen,3)$ in $KB(D_{scp},H_{scp})$:

$$\delta_1 = \{happens(steal,0)\},$$
$$\delta_2 = \{happens(steal,1)\},$$
$$\delta_3 = \{happens(steal,2)\}$$

In order to incorporate N into $KB(D,H)$, if there is a possible cause δ for N in $KB(D,H)$, then we can simply add δ into the knowledge base to have a new knowledge base. Now we have to question about whether there is always a possible cause δ for N in $KB(D,H)$. The answer turns out to be negative.

Proposition 5.9 *Let D be a domain description, and H a history. Let N be a set of literals. It does not hold that there is always a possible cause for N in $KB(D,H)$.*

For incorporation of N into $KB(D,H)$, when there is no possible cause for N in $KB(D,H)$, we will simply discard N because N cannot be explained.

Definition 5.10 *Given a knowledge base $KB(D,H)$, and a set of new facts N, all the possible new knowledge bases $KB(D,H) \oplus N$ are defined to be*

$$\{KB(D,H) \dot{+} \delta :\ \delta \in KB(D,H) \uparrow N\}$$

The operator $\oplus$ is called the PCA knowledge incorporation operator.

For example, $KB(D_{scp}, H_{scp}) \oplus holds(stolen, 3)$ includes, in the Stolen Car Problem,

$$KB(D_{scp}, \{(steal, 0)\}),\ KB(D_{scp}, \{(steal, 1)\}),\ \text{and}\ KB(D_{scp}, \{(steal, 2)\}).$$

Note that the new knowledge N is not directly added to the knowledge base. This can be problematic in general. As discussed in its long version, there is a simple way to modify the above definition so that both new knowledge and its possible causes are incorporated, described as follows. For each fact $holds(F, T)$ and $\neg holds(G, T)$ in N, we add the following two constraints, respectively:

$$\begin{aligned} false &\leftarrow not\ holds(F, T) \\ false &\leftarrow holds(G, T) \end{aligned}$$

Then, N is kept permanently in the knowledge base.

The PCA knowledge incorporation operator can also be used to evaluate counterfactual queries: if F_1 were true at time T_1, would F_2 be true at time T_2? This amounts to evaluate the query "$? - holds(F_2, T_2)$" in $KB(D, H) \oplus holds(F_1, T_1)$. The details are omitted.

5.7 Computational considerations

The critical step in the PCA approach is the computation of the possible causes. In this section we show how to compute them with abductive logic programming and how to improve computational efficiency.

First we introduce a new predicate $occurs(A, T)$, for action A and time T, and add a new programming rule to $KB(D, H)$:

$$happens(A, T) \leftarrow occurs(A, T) \tag{1}$$

Now we define the abducibles to be $initially(F)$ and $occurs(A, T)$. The above rule means that if action A is abduced to occur at time T, it happens at T. As said before, this paper does not consider concurrent actions. Thus we add the following rule:

$$false \leftarrow A_1 \neq A_2, happens(A_1, T), happens(A_2, T) \tag{2}$$

The definitions for $A1 \neq A2$ should also be added such that every two syntactically different actions are not equal. However, later we will find that the above rule is redundant and can be removed for efficiency improvement. Let $KB_{occurs}(D, H) = KB(D, H) \cup \{(1), (2)\}$. It is easy to see that KB_{occurs} is still acyclic. It can also be shown that possible causes for N become abductive answers to N in $KB_{occurs}(D, H)$.

Proposition 5.11 *$\delta \in KB(D, H) \uparrow N$ iff $\{initially(F) : initially(F) \in \delta\}$ $\cup\{occurs(A, T) : happens(A, T) \in \delta\}$ is an abductive answer to N in the abductive logic program $KB_{occurs}(D, H)$.*

In what follows we develop some techniques to improve the search efficiency.

Proposition 5.12 *Let δ be any abductive answer to $holds(F_n, T_n)$ in the program $KB_{occurs}(D, H)$. Then, for any $occurs(A, T) \in \delta$ we have $T \prec T_n$.*

Thus it is not necessary to consider predicate instances $occurs(A, T)$ for $T \not\prec T_n$. The second observation is that we have assumed that no concurrent actions have happened in histories. Thus, if $(A, T) \in H$, we need not consider $occurs(B, T)$ for any other action B.

Proposition 5.13 *Let δ be any abductive answer to N in $KB_{occurs}(D, H)$. If $occurs(a, t) \in \delta$, then there is no other action b such that $(b, t) \in H$.*

The third observation is about the consistency check. $KB(D, H) \dot{+} \delta$, by Definition 5.8, needs to be consistent for δ to be a possible cause. To check the consistency of a knowledge base is very expensive. In the following we give a sufficient and cheap technique to check the consistency.

Proposition 5.14 *Let D be an e-consistent domain description. If D is not consistent, then there are two value propositions* **initially** F *and* **initially** G *such that $F = \neg G$ or $G = \neg F$.*

By the above proposition, the consistency check can be done in the following way: For each fluent F, we add a new programming rule to $KB(D, H)$:

$$false \leftarrow holds(F, init), not\ holds(F, init)$$

All the examples of this paper and a few more benchmark examples have been experimented with the latest version of the REVISE system [9].

6 Concurrent Action

6.1 Motivation

In reality, occurrences of many actions overlap in time, which complicates temporal prediction and explanation in AI. For definite goals some actions may be planned to be carried out at the same time in order to save time, or to decrease production cost, or for other context-dependent purposes.

A concurrent action is understood as a finite nonempty set of atomic actions, which happen at the same time. A nonempty subset of a concurrent action is called a subaction of the concurrent action. For example, opening a door and switching on a light at the same time is a concurrent action consisting of two atomic actions: opening a door and switching on a light. In common sense, the effect of a concurrent action is the aggregation of those of its subactions. For example, after one opens a door and switches on a light at the same time, the door is open and the light is on, which is the aggregation of the effects of opening the door and switching on the light.

The effect relationship between an action and its subactions was first discussed by Gelfond, Lifschitz and Robinov in the framework of the situation calculus [15], and further explored by Lin and Shoham [23], and Baral and Gelfond [7]. According to [23, 7], essentially, a concurrent action *usually* inherits effects of its subactions. There is no problem with such a default when a concurrent action does not include atomic actions with conflicting effects. When a concurrent action includes atomic actions with conflicting effects, they may give intuitively undesired results. According to [23], the epistemological completeness implies that in any situation one can always decide whether a fluent is true or not. Thus, if the door is initially closed, then the door will still be closed after $\{Open, Close\}$ is done; if the door is initially open, then the door will still be open after $\{Open, Close\}$ is done. According to [7], after $\{Open, Close\}$ is done, the resulting situation does not exist and $\{Open, Close\}$ is not executable.

In this paper we will introduce a cautious semantics to concurrent actions. When subactions of a concurrent action do not have conflicting effects, it can be proved that our semantics coincides with Lin-Shoham's and Baral-Gelfond's. When two subactions of a concurrent action have conflicting effects, the truth value of all fluents affected by the two subactions will be left undefined. For example, the fluent denoting the status of the door will be left undefined after $\{Open, Close\}$ is done.

6.2 Domain Descriptions

Syntax of $\mathcal{AC}$ The syntax of $\mathcal{AC}$is similar $\mathcal{A}$ except that an action expression is a finite set of atomic actions. Thus, a domain description in $\mathcal{AC}$is a set of effect assertions of the form A **causes** F **if** $P_1, \ldots, P_n$, and value assertions of the form F **after** $A_1, \ldots, A_m$, where $A, A_1, \ldots, A_m$ may be a set of atomic actions.

Example 6.1 (Mary's Soup [15]) *Whenever Mary lifts the bowl with only one hand, she will spill the soup. But when she lifts the bowl with two hands, she will not spill the soup. Let LL and LR to denote lifting the bowl with the left hand and the right hand, respectively. Then, the domain can be described as follows:*

$$D_{soup} = \left\{ \begin{array}{l} \textbf{initially} \ \ \neg Spilled. \\ LL \ \textbf{causes} \ \ Spilled. \\ LR \ \textbf{causes} \ \ Spilled. \\ \{LL, LR\} \ \textbf{causes} \ \ \neg Spilled. \end{array} \right\}$$

Semantics of $\mathcal{AC}$ The semantics of $\mathcal{AC}$is defined by using states and transitions.

A *state* σ is a pair of sets of fluent names $\langle \sigma^+, \sigma^- \rangle$ such that σ^+ and σ^- are disjoint, i.e., $\sigma^+ \cap \sigma^- = \emptyset$. Given a fluent name F and a state σ, we say that F holds in σ if $F \in \sigma^+$, F does not hold in σ if $F \in \sigma^-$, and it is not known whether F holds in σ otherwise. Given a fluent name F, we also say that $\neg F$ holds in σ if F does not hold in σ. Sometimes we also say that F is true in σ if $F \in \sigma^+$, F is false in σ if $F \in \sigma^-$, and F is undefined in σ otherwise.

A *transition function* Φ is a mapping from the set of pairs (A, σ), where A is an action expression and σ is a state, into the set of states.

A *structure* is a pair (σ_0, Φ), where σ_0 is a state, called the *initial state* of the structure, and Φ is a transition function. For any structure $M = (\sigma_0, \Phi)$ and any sequence of action expressions $A_1; \ldots; A_m$ in M, by $\Phi(A_1; \ldots; A_m, \sigma_0)$ we denote the state $\Phi(A_m,\ \Phi(A_{m-1},\ \ldots,\ \Phi(A_1, \sigma_0) \ldots))$. A v-assertion of the form F **after** $A_1, \ldots, A_m$ is satisfied in a structure $M = (\sigma_0, \Phi)$ iff F holds in the state $\Phi(A_1; \ldots; A_m, \sigma_0)$.

We say that the execution of an action A in a state σ *immediately initiates* a fluent expression F if there is an e-proposition A **causes** F **if** $P_1, \ldots, P_m$ in the domain description such that for each $1 \leq i \leq m$, P_i holds in σ. Moreover, we say that the execution of A in σ *initiates* a fluent expression F if A immediately initiates F, or there is a $B \subseteq A$ such that execution of B in σ *immediately initiates* F and there is no C such that $B \subset C \subseteq A$, where execution of C in σ immediately initiates $\neg F$.

Let F be positive. When A initiates $\neg F$, we will say that A terminates F. We define two set-ranged functions as follows:

$$
\begin{aligned}
Initiate(A, \sigma) &= \{F \ :\ F \in \Sigma_f \text{ and } A \text{ initiates } F \text{ in } \sigma\} \\
Terminate(A, \sigma) &= \{F \ :\ F \in \Sigma_f \text{ and } A \text{ initiates } \neg F \text{ in } \sigma\}
\end{aligned}
$$

Note that $Initiate(A, \sigma)$ and $Terminate(A, \sigma)$ are not necessarily disjoint. We also need a set-ranged function $Cause(F, \sigma, A)$, (for fluent name F, state σ, and action expression A), which is defined to contain and only contain all the *set-inclusion minimal* subactions B of A satisfying: B immediately initiates F (or $\neg F$, resp.) in σ and there is no C such that $B \subset C \subseteq A$, where C immediately initiates $\neg F$ (or F, resp.) in σ. If $B \in Cause(F, \sigma, A)$, we will say that B is a cause for the change in truth values of the fluent name F in σ when A is done. The truth value of the fluent name F may change in two ways: from $true$ to $false$ and from $false$ to $true$. Note that the subaction B is set-inclusion minimal among all subactions which satisfy one of the two conditions above.

For later purpose we also need one more auxiliary functions:

$$\Delta(A, \sigma) = \bigcup_{B \in W} (Initiate(B, \sigma) \cup Terminate(B, \sigma))$$

where

$$W = \bigcup_{F \in Initiate(A,\sigma) \cap Terminate(A,\sigma)} Cause(F, \sigma, A)$$

Intuitively speaking, $\Delta(A, \sigma)$ denotes those fluents influenced by subactions of A which have conflicting effects. All of the fluents influenced by these subactions will be made undefined in the new situation resulting from doing A.

Definition 6.2 (Model) *A structure* (σ_0, Φ) *is a* model *of a domain description D iff*

– *Every v-proposition of D is satisfied in* (σ_0, Φ).

– *For every action expression A and every state σ,*

$$\Phi(A,\sigma) = \langle S^+, \quad S^- \rangle$$

where

$$S^+ = (\sigma^+ \cup Initiate(A,\sigma)) \setminus Terminate(A,\sigma) \setminus \Delta(A,\sigma)$$
$$S^- = (\sigma^- \cup Terminate(A,\sigma)) \setminus Initiate(A,\sigma) \setminus \Delta(A,\sigma)$$

A domain description is consistent *if it has a model. A domain description is* complete *if it has exactly one model. We will use $Mod(D)$ to denote the set of all models of D.*

Definition 6.3 (Entailment) *A v-proposition is* entailed *by a domain description D if it is satisfied in every model of D.*

All of D_{YSP}, D_{door}, and D_{soup} defined before are complete since they have only one model, respectively.

6.3 Translation into Abductive Logic Programs

In order to automate temporal reasoning in $\mathcal{A}$, now we present a sound and complete translation from domain descriptions in $\mathcal{AC}$to abductive logic programs. The familiarity with abductive logic programming is assumed. imm stands for "immediately" in $imm\_initiates(A,F,S)$ and $imm\_terminates(A,F,S)$. In the translation, all the predicates are either self-explanatory or mirror the definitions of Subsection 6.2. The symbol *not* in logic programs denotes the negation-as-failure operator.

Let D be a domain description in $\mathcal{AC}$. The translation πD consists of a normal logic program and a set of constraints, defined as follows:

1. Auxiliary predicates about subactions:
   Assume given standard rules for set-theoretical predicates $member(A,S)$, $subseteq(S_1,S_2)$. Sets can be represented by lists. Using these predicates we can define $subacteq(S_1,S_2)$ and $subact(S_1,S_2)$ about subactions in logic programming rules.
2. Initialization:

$$is\_true(F,s_0) \leftarrow initially\_true(F). \tag{3}$$
$$is\_false(F,s_0) \leftarrow initially\_false(F). \tag{4}$$

   where s_0 is a new symbol to denote the initial situation. $initially\_true(F)$ and $initially\_false(F)$ are taken to be abducible predicates. If a fluent name F is abduced to be true (false, resp.) initially, then it is true (false, resp.) in the initial situation s_0.

3. Auxiliary Predicates:

   These predicates will be used to define two main predicates $is\_true(F,S)$ and $is\_false(F,S)$, indicating whether fluent F is true or false in situation S.

$$
\begin{aligned}
&initiates(A,F,S) \leftarrow imm\_initiates(A,F,S).\\
&initiates(A,F,S) \leftarrow subacteq(B,A), imm\_initiates(B,F,S),\\
&\qquad\qquad not\ clip\_initiates(F,B,A,S).\\[1ex]
&terminates(A,F,S) \leftarrow imm\_terminates(A,F,S).\\
&terminates(A,F,S) \leftarrow subacteq(B,A), imm\_terminates(B,F,S),\\
&\qquad\qquad not\ clip\_terminates(F,B,A,S).\\[1ex]
&causes(F,S,A,B) \leftarrow subacteq(B,A), imm\_initiates(B,F,S),\\
&\qquad\qquad not\ clip\_initiates(F,B,A,S),\\
&\qquad\qquad not\ clip\_cause1(F,B,A,S).\\
&causes(F,S,A,B) \leftarrow subacteq(B,A), imm\_terminates(B,F,S),\\
&\qquad\qquad not\ clip\_terminates(F,B,A,S),\\
&\qquad\qquad not\ clip\_cause2(F,B,A,S).\\[1ex]
&delta(A,S,F) \leftarrow initiates(A,G,S), terminates(A,G,S),\\
&\qquad\qquad causes(G,S,A,B), initiates(B,F,S).\\
&delta(A,S,F) \leftarrow initiates(A,G,S), terminates(A,G,S),\\
&\qquad\qquad causes(G,S,A,B), terminates(B,F,S).\\[1ex]
&clip\_initiates(F,B,A,S) \leftarrow subact(B,C), subacteq(C,A),\\
&\qquad\qquad imm\_terminates(C,F,S).\\[1ex]
&clip\_terminates(F,B,A,S) \leftarrow subact(B,C), subacteq(C,A),\\
&\qquad\qquad imm\_initiates(C,F,S).\\[1ex]
&clip\_cause1(F,B,A,S) \leftarrow subacteq(B,A), subact(C,B),\\
&\qquad\qquad imm\_initiates(C,F,S),\\
&\qquad\qquad not\ clip\_initiates(F,C,A,S).\\[1ex]
&clip\_cause2(F,B,A,S) \leftarrow subacteq(B,A), subact(C,B),\\
&\qquad\qquad imm\_terminates(C,F,S),\\
&\qquad\qquad not\ clip\_terminates(F,C,A,S).
\end{aligned}
$$

4. Main Predicates:

   The following predicates are used to determine whether a fluent is true or false in a situation.

$$\begin{aligned} is\_true(F, result(A, S)) &\leftarrow is\_true(F, S), not\ terminates(A, F, S), \\ &\quad\ not\ delta(A, S, F). \\ is\_true(F, result(A, S)) &\leftarrow initiates(A, F, S), not\ terminates(A, F, S), \\ &\quad\ not\ delta(A, S, F). \end{aligned}$$

$$\begin{aligned} is\_false(F, result(A, S)) &\leftarrow is\_false(F, S), not\ initiates(A, F, S), \\ &\quad\ not\ delta(A, S, F). \\ is\_false(F, result(A, S)) &\leftarrow terminates(A, F, S), not\ initiates(A, F, S), \\ &\quad\ not\ delta(A, S, F). \end{aligned}$$

5. Domain-Specific Predicates
The syntax and semantics of the following predicates depend on domain descriptions. Let F be a fluent name, i.e., $F \in \Sigma_f$. Then, we write $Holds(F, S)$ to stand for $is\_true(F, S)$ and write $Holds(\neg F, S)$ for $is\_false(F, S)$.
   - For each effect proposition a **causes** f **if** $p_1, \ldots, p_n$ in D, where f is positive, we have a logic programming rule:

$$\begin{aligned} &imm\_initiates(a, f, S) \leftarrow \\ &\qquad Holds(p_1, S), \ldots, Holds(p_n, S). \end{aligned}$$

   - For each effect proposition a **causes** $\neg f$ **if** $p_1, \ldots, p_n$, where f is positive, we have a logic programming rule:

$$\begin{aligned} &imm\_terminates(a, f, S) \leftarrow \\ &\qquad Holds(p_1, S), \ldots, Holds(p_n, S). \end{aligned}$$

The integrity constraints, denoted by IC_D, are defined as follows: For each value proposition F **after** $A_1; \ldots; A_m$, we have:

$$Holds(F, result(A_1; \ldots; A_m, s_0)) \tag{5}$$

As we will use the predicate completion semantics, the above integrity constraint can be equivalently transformed into

$$false \leftarrow \text{not } Holds(F, result(A_1; \ldots; A_m, s_0)).$$

Note that we cannot abduce a fluent to be both true and false. For this purpose we add the following domain-independent constraint:

$$false \leftarrow is\_true(F, S_0), is\_false(F, S_0). \tag{6}$$

The literal $false$ is always interpreted as logical falsity, and all rules with $false$ as heads function as integrity constraints. The predicates $initially\_true(F)$ and $initially\_false(F)$ are taken as abducible predicates used to capture the incomplete knowledge about the initial situation. The semantics of πD is defined to be the union of the integrity constraints, the Clark Equality Theory, and the first-order theory obtained by completing all the non-abducible predicates [8]. The definition for abducible predicates $initially\_true(F)$ and $initially\_false(F)$ are left open. We will write $COMP(\pi D)$ to denote the semantics of of πD. The following two results justify the above semantics definition.

Proposition 6.4 *πD is an acyclic program with first-order constraints in the sense of [5].*

Corollary 6.5 *For the abductive logic program πD, $COMP(\pi D)$ coincides with its generalized stable model semantics [17] and generalized well-founded model semantics [27].*

The following two technical results are proved in the long version of this paper.

Theorem 6.6 (Soundness) *Let D be any domain description. For any value proposition Q, if $COMP(\pi D) \models \pi Q$, then D entails Q.*

Theorem 6.7 (Completeness) *Let D be a domain description. For any value proposition Q, if D entails Q, then $COMP(\pi D) \models \pi Q$.*

Since our translation is both sound and complete, we can reduce temporal reasoning in $\mathcal{AC}$to abductive query evaluaiton in logic program πD. The work reported in this paper has been experimented with the latest version of the REVISE system [9]. The use of the REVISE system is not essential, and other abductive query evaluaiton procedures for normal logic programs such as SLD-NFA [11] and that in [30] should also work. If for every fluent name F, one and only one of **initially** F and **initially** $\neg F$ appears in a domain description, then PROLOG can be used for temporal projection after (3), (4), and (6) are removed and the integrity constraitns are represented only in the form of (5). For example, temporal reasoning in the Yale Shooting domain D_{YSP} can be done in PROLOG. Furthermore, if we introduce more propositions, say observation propositions, we can abductively diagnose action domains when there! is a discrepancy between predicted propositions and observed propositions. For the sake of space limitation, we will not go into deeper discussions.

6.4 An Unsolved Problem

Now let's condider the following so-called Poor Street Musician Problem: Suppose there is a poor street musician with a violin and a money bag. While he plays the violin in the street, the by-passers may throw some money into the money bag. It is often the case that two or more by-passers may throw their money at the same time. For simplicity, we write $Money(x)$ to denote that he has x dollars in the bag, and $Gives(x)$ to denote that a by-passer gives x dollars to the musician. Then, the domain can be described as something like this:

$$\begin{array}{c} \textbf{initially } \ Money(0) \\ Gives(x) \textbf{ causes } \ Money(x+y) \textbf{ if } \ Money(y) \\ Gives(x) \textbf{ causes } \ \neg Money(y) \textbf{ if } \ Money(y) \end{array}$$

Then, what's the effect of the concurrent action $\{Gives(1), Gives(2)\}$. The intuitive answer should be that there are three more dollars in the bag. However, it can be found that neither of the above three inheritance-based semantics works in this example.

An alternative semantics would be expected to be based on so-called *interleaving trace* of concurrent actions [3].

However, the interleaving trace based semantics does not work well if one slightly modifies Gelfond-Lifscitz-Robinov's Mary's Soup example: Suppose that Mary wants to lift a bowl of soup to serve a guest. Whenever she lifts the bowl with only one hand, she will spill the soup to the floor. But when she lifts the bowl with two hands, she will not spill the soup. What will be the effects of $\{sing, liftwithleft-hand, liftwithrighthand\}$? It can be seen that the interleaving- trace-based semantics does not work for this example.

Now the problem is whether there is any elegant semantics for concurrent actions which can uniformly deal with both Poor Street Musician Problem and Modified Mary's Soup Problem?

7 Concluding Remarks

In this paper we have addressed three problems related to development of intelligent agents: refinement of action theories, knowledge assimilation, and concurrent actions. All the three problems can be solved in part by using abductive logic programs. The declarative semantics and efficient execution of abductive logic programs have made it possible to develop a formal, provably correct with respect to action theories in a high-level action description language ($\mathcal{A}$ family language), and yet computational methodology for reasoning about actions. In fact, abductive logic programmming can also be applied in many other occasions: intention recognitions of other agents, diagnosis, planning, etc., which are some of our current research topics in progress.

Acknowledgement

This work was partially supported by JNICT of Portugal under MENTAL project PRAXIS 2/2.1/TIT/1593/95 and PRAXIS XXI/BPD/4165/94.

References

1. C. Alchorrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50(2):510 – 530, 1985.
2. J. J. Alferes, C. V. Damásio, and L. M. Pereira. SLX - a top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *International Logic Programming Symposium.* MIT Press, 1994.
3. J. J. Alferes, R. Li, and L. M. Pereira. Non-atomic actions in the situation calculus. In N. Mamede and C. Pinto-Ferreira, editors, *Progress in Artificial Intelligence, LNAI 990*, pages 273–284. Springer, 1995.
4. J. J. Alferes and L.M. Pereira. *Reasoning with Logic Programming, LNAI 1111.* Springer, 1996.

5. K. R. Apt and M. Bezem. Acyclic programs. In *Proc. of ICLP 90*, pages 579–597. MIT Press, 1990.
6. A. B. Baker. Nonmonotonic reasoning in the framework of situation calculus. *Artificial Intelligence*, 49:5–23, 1991.
7. C. Baral and M. Gelfond. Representing concurrent actions in extended logic programming. In *Proc. of IJCAI'93*, pages 866–871. Morgan Kaufmann, 1993.
8. L. Console, D. T. Dupré, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.
9. C. V. Damásio, L.M. Pereira, and W. Nejdl. Revise: An extended logic programming system for revising knowledge bases. In *Proc. of KR'94*, 1994.
10. M. Denecker. Knowledge representation and reasoning in incomplete logic programming. Ph.D. thesis, Department of Computer Science, K.U.Leuven, 1993.
11. M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for normal abductive programs. In K. R. Apt, editor, *Logic Programming: Proc. of 1992 Int'l Joint Conference and Symposium*, pages 686–700. MIT Press, 1992.
12. M. Denecker and D. de Schreye. Representing incomplete knowledge in abductive logic programming. In *Logic Programming: Proc. of the 1993 Int'l Symposium*, pages 147–163. MIT Press, 1993.
13. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proc. of 5th Logic Programming Symposium*, pages 1070–1080. MIT Press, 1988.
14. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.
15. M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In R. Moore, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 167–179. 1991.
16. M. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(1):35 – 79, 1986.
17. A.C. Kakas and P. Mancarella. Generalized stable models: A semantics for abduction. In *Proc. of ECAI'90*, 1990.
18. G.N. Kartha. Soundness and completeness theorems for three formalizations of action. In *Proc. IJCAI93*, pages 712–718. MIT Press, 1993.
19. R.A. Kowalski and F. Sadri. The situation calculus and event calculus compared. In *Proc. of ILPS 94*, pages 539–553. MIT Press, 1994.
20. R. Li and L.M. Pereira. Knowledge assimilation in domains of actions: A possible causes approach. *Journal of Applied Non-Classical Logic*, Special issue on Inconsistency Handling in Knowledge Systems (guest editor: G. Wagner), 1997.
21. R. Li and L.M. Pereira. Knowledge-based situated agents among us. In J. P. Muller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III – Proc. of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96), LNAI 1193*, pages 375–389. Springer, 1997.
22. F. Lin and Y. Shoham. Provably correct theories of actions: preliminary report. In *Proc. of AAAI-91*, 1991.
23. F. Lin and Y. Shoham. Concurrent actions in the situation calculus. In *Proc. of AAAI-92*, pages 590–595, 1992.
24. J. McCarthy and P.J. Hayes. Some philosophical problems from the stand-point of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502, Edinburgh, 1969.
25. E. P. D. Pednault. Adl: Exploring the middle ground between strips and the situation calculus. In R. J. Brachman, H. Levesque, and R. Reiter, editors, *Proc. of KR'89*, pages 324–332. Morgan Kaufmann Publishers, Inc., 1989.

26. L. M. Pereira and J. J. Alferes. Well-founded semantics for logic programs with explicit negation. In B. Neumann, editor, *Proc. of ECAI'92*, pages 102–106. John Wiley & Sons, 1992.
27. L. M. Pereira, J. J. Alferes, and J. N. Aparício. Nonmonotonic reasoning with well founded semantics. In K. Furukawa, editor, *Proc. of 8th ICLP*, pages 475–489. MIT Press, 1991.
28. R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.
29. E. Sandewall. *Features and Fluents: The Representation of Knowledge about Dynamic Systems, Vol. 1.* Oxford University Press, 1994.
30. K. Satoh and N. Iwayama. A query evaluation method for abductive logic programming. In K. R. Apt, editor, *Logic Programming: Proc. of 1992 Int'l Joint Conference and Symposium*, pages 671–685, 1992.
31. A. Van Gelder, K. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:620–650, 1991.
32. M. Winslett. Reasoning about action using a possible models approach. In *AAAI'88*, pages 89 – 93, 1988.