Reasoning about Concurrent Actions and Observations

Renwei Li and **Luís Moniz Pereira**
Department of Computer Science
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal
{renwei | lmp}@di.fct.unl.pt

Abstract In this paper we present an experiment by using abductive logic programming to reason about concurrent actions and observations. Technically, we extend Gelfond and Lifschitz' action description language $\mathcal{A}$ with concurrent actions and observation propositions to describe the ideal behaviour of domains of (concurrent) actions and practically observed behaviour, respectively, without requiring that the practically observed behaviour of a domain of actions be consistent with its ideal behaviour. We present a translation from domain descriptions and observations in the new action language to abductive normal logic programs. The translation is shown to be both sound and complete. From the standpoint of diagnosis, in particular, we discuss the temporal explanation of inferring actions from fluent changes at two different levels, namely, at the domain description level and at the abductive logic programming level. The method is applicable to the temporal projection problem with incomplete information, as well as to the temporal explanation of inferring actions from fluent changes.

1 Introduction

The main tasks involved in reasoning about actions and changes are as follows: Given a domain description with fluents holding and actions happening at certain times or states, we want to reason forward and backward in time, inferring actions from changes of fluents and inferring changes of fluents from actions. This paper is in line with the work related to the action description language $\mathcal{A}$. Now let's consider the Stolen Car Problem (SCP) [8]. Let the SCP domain have one fluent name *Stolen* and two action names *Wait* and *Steal*. Then, its description D_1 in $\mathcal{A}$ is as follows:

initially $\neg Stolen$
Stolen **after** $Wait; Wait; Wait$
Steal **causes** *Stolen*

It can be shown that D_1 is inconsistent in $\mathcal{A}$, and thus has no model. However, it seems that the above three propositions roughly correspond to a stolen-car story.

There is nothing wrong with the above story, and we should have a model for the SCP domain. Our stance is that what is wrong with the domain description D_1 is that we only have incomplete knowledge about what has happened in the past. The action *Steal* has happened *at the same time* as one of the three *Wait* actions. Thus, at least one of the situations $Result(Wait, S_0)$, $Result(Wait; Wait, S_0)$, $Result(Wait; Wait; Wait, S_0)$ does not correctly represent the real situation.

In order to solve the problems such as SCP, we need to extend the action language $\mathcal{A}$ with concurrent actions, and to express incomplete knowledge about the past history. On the other hand, extension of $\mathcal{A}$ with concurrent actions has its own right in domain descriptions as argued in [2]. In this paper we will extend $\mathcal{A}$ with concurrent actions, but in a different way from that of [2] in the semantics and the translation into logic programs. For representation of incomplete knowledge about the past history we need to extend the action language $\mathcal{A}$. We will introduce a new kind of propositions, called *observation propositions*, whose syntax is as follows: **observed** F **after** $A_1; \ldots; A_m$. The observation proposition of the above form simply means that F is true after performance of actions $A_1; A_2; \ldots; A_m$ are observed. Because of the incomplete observation, some other actions may have also been done while actions $A_1, A_2, \ldots, A_m$ are done.

The rest of this paper is organized as follows. In Section 3 we give a modification of $\mathcal{A}$ and its semantics and enrich it with concurrent actions and observation propositions. In Section 4 we present a translation from domain descriptions in the new action language into abductive logic programs. In Section 5 we discuss the temporal explanation from the standpoints of diagnosis. The first part of the paper concerning concurrent actions has been reported in [10]. All the proofs of the technical results are omitted, but can be found in its longer version.

2 Concurrent actions

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 has been explored by Lin and Shoham [11], and Baral and Gelfond [2].

Let *Open* and *Close* to denote two atomic actions: to make sure to open and close the door, respectively. In common sense, *Open* and *Close* have conflicting effects. Now consider a concurrent action $\{Open, Close\}$. How should we evaluate and deal with concurrent actions with conflicting effects? Lin-Shoham and Baral-Gelfond give two different methods.

In Lin and Shoham's solution [11], it is required to be able to derive whether or not the door is closed in the next situation in order to guarantee the epistemological completeness. 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. This is true when both actions use exactly the same force according to mechanics. But it is not the case in common sense, and it does not seem to be well justified that we could predict a complete description of the resulting situation after the concurrent action $\{Open, Close\}$ is done.

In Baral and Gelfond's solution [2], after $\{Open, Close\}$ is done, the resulting situation does not exist and $\{Open, Close\}$ is not executable. Baral and Gelfond define the state (situation) transition by doing actions as a partial mapping. For those concurrent actions with conflicting effects such as $\{Open, Close\}$, the state transition is not defined. By [2], any successive actions are not executable after $\{Open, Close\}$ is done. Now assume we have an additional action, *Switch*, which denotes switching a light. According to [2] the action sequence $\{Open, Close\}; Switch$ is unexecutable. In common sense, the status of the light should change, although the status of the door is not known. Now consider another concurrent action $\{Open, Close, Switch\}$. According to [2] the concurrent action $\{Open, Close, Switch\}$ is not executable, and the resulting situation is not defined. Although *Open* and *Close* have conflicting effects, we can at least infer that the status of the light will be changed after $\{Open, Close, Switch\}$ is done.

In this paper we will give a new semantics to concurrent actions. When subactions of a concurrent action do not have conflicting effects, 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. In our new semantics, we can infer, for example, the door will be closed after the sequence of actions $\{Open, Close\}; Close$ is done, the status of the light will be changed after $\{Open, Close, Switch\}$ or $\{Open, Close\}; Switch$ is done. Reasonign about effects of (concurrent) actions is automated with abductive logic programming, and temporal reasoning amounts to abductive query evaluation.

3 An action description language

Given a domain, we will make a distinction between its description and its observation. A domain description tells how the domain should behave, while an observation of it tells some incomplete past history. In this section we will only consider the domain description part of $\mathcal{A}_{CO}$.

3.1 Domain descriptions

We begin with the alphabet Σ of a domain, which consists of two disjoint non-empty sets of symbols Σ_f and Σ_a, called *fluent names* and *atomic action names*, respectively.

We will often use self-explanatory fluent names and atomic action names when no confusion arises. A ***fluent expression*** is 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*. An *action expression* is a non-empty finite set of atomic action names. We say that an action expression A is a *subaction expression* (*proper subaction expression*) of B iff $A \subseteq B$ ($A \subset B$). If A is a subaction expression of B, we often say that B contains A. We will simply write A for $\{A\}$ if A is atomic. Fluent expressions and action expressions will simply be called fluents and actions, respectively, if no confusion arises.

In $\mathcal{A}_{CO}$ there are two kinds of propositions: value propositions and effect propositions, simply called v-propositions and e-propositions. A v-proposition is a statement of the form

$$F \textbf{ after } A_1;\ldots;A_m \tag{1}$$

where F is a fluent expression and $A_1,\ldots,A_m (m \geq 0)$ are action expressions. If $m = 0$, then we will write it as **initially** F. An e-proposition is a statement of the form

$$A \textbf{ causes } F \textbf{ if } P_1,\ldots,P_n \tag{2}$$

where A is an action expression, and each of $F, P_1,\ldots,P_n (n \geq 0)$ is a fluent expression. If $n = 0$, then we will write it as A **causes** F.

A domain description is a set of v-propositions and e-propositions. For example, Hanks-McDermott's Yale Shooting domain is described as follows: D_{YSP} = { **initially** $\neg Loaded$. **initially** $Alive$. $Shoot$ **causes** $\neg Alive$ **if** $Loaded$. $Shoot$ **causes** $\neg Loaded$. $Load$ **causes** $Loaded$}.

3.2 Semantics

The semantics of $\mathcal{A}_{CO}$ 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. 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 actions $A_1;\ldots;A_m$ 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-proposition of the form (1) 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, the set $\Delta(A,\sigma)$ 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 3.1 (Model) *A structure (σ_0, Φ) is a* model *of a domain description D iff (i) Every v-proposition of D is satisfied in (σ_0, Φ). (ii) For every action expression A and every state σ, $\Phi(A,\sigma) = \langle S^+, S^- \rangle$ where $S^+ = (\sigma^+ \cup Initiate(A,\sigma)) \setminus Terminate(A,\sigma) \setminus \Delta(A,\sigma)$, and $S^- = (\sigma^- \cup Terminate(A,\sigma)) \setminus Initiate(A,\sigma) \setminus \Delta(A,\sigma)$. 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 3.2 (Entailment) *A v-proposition is* entailed *by a domain description D if it is satisfied in every model of D.*

4 Translation into logic programs

An abductive normal logic programming framework [6] is a triple (P, A, I), where P is a set of normal logic programming rules, A a set of abducibles, and I a set of integrity constraints. The semantics of an abductive normal logic program is defined to be the union of the integrity constraints and the first-order theory by completing the non-abducible predicates together with Clark's Equality Theory (CET). For all

the acyclic logic programs [1], the predicate completion semantics coincide with other semantics such as the stable model semantics and the well-founded model semantics. In the case of abductive logic programs, the semantical coincidence still holds, as shown by Denecker [4]. In the following translation, all the predicates will be self-explanatory, where *imm* stands for *immediate*. Let D be a domain description in $\mathcal{AC}$. The translation πD is defined as follows:

1. Auxiliary predicates about subactions: Assume that we have standard rules for set-related predicates $Subseteq(S_1, S_2)$ and $Member(A, S)$, by which we can define $subacteq(S_1, S_2)$ and $subact(S_1, S_2)$ about subactions.

2. Initialization: Let s_0 be a new symbol to denote the initial situation.

$$\begin{aligned} is\_true(F, s_0) &\leftarrow initially\_true(F). \\ is\_false(F, s_0) &\leftarrow initially\_false(F). \end{aligned}$$

3. Auxiliary Predicates: These auxiliary predicates will be used to define the two main predicates $is\_true(F, S)$ and $is\_false(F, S)$, which indicate whether fluent F is true or false in situation S.

$$\begin{aligned}
initiates(A,F,S) &\leftarrow imm_initiates(A,F,S) \\
initiates(A,F,S) &\leftarrow subacteq(B,A), imm_initiates(B,F,S), \\
&\quad\ not\ clip_initiates(F,B,A,S) \\
terminates(A,F,S) &\leftarrow imm_terminates(A,F,S) \\
terminates(A,F,S) &\leftarrow subacteq(B,A), imm_terminates(B,F,S), \\
&\quad\ not\ clip_terminates(F,B,A,S) \\
causes(F,S,A,B) &\leftarrow subacteq(B,A), imm_initiates(B,F,S), \\
&\quad\ not\ clip_initiates(F,B,A,S), \\
&\quad\ not\ clip_Cause1(F,B,A,S) \\
causes(F,S,A,B) &\leftarrow subacteq(B,A), imm_terminates(B,F,S), \\
&\quad\ not\ clip_terminates(F,B,A,S), \\
&\quad\ not\ clip_Cause2(F,B,A,S) \\
delta(A,S,F) &\leftarrow initiates(A,G,S), terminates(A,G,S), \\
&\quad\ causes(G,S,A,B), initiates(B,F,S) \\
delta(A,S,F) &\leftarrow initiates(A,G,S), terminates(A,G,S), \\
&\quad\ causes(G,S,A,B), terminates(B,F,S)
\end{aligned}$$

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

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

5. Domain-Specific Predicates

   The syntax and semantics of the following predicates depend on domain descriptions. Let $F \in \Sigma_f$ be a fluent name. We write $holds(F, S)$ and $holds(\neg F, S)$ to stand for $is\_true(F, S)$, and $is\_false(F, S)$, respectively.

   - 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:

$$imm\_initiates(a, f, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S).$$

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

$$imm\_terminates(a, f, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S).$$

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

$$holds(F, result(A_1; \ldots; A_m, s_0)) \tag{3}$$

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

$$false \leftarrow not\ holds(F, result(A_1; \ldots; A_m, s_0)).$$

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

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

The literal $false$ is always interpreted as logical falsity, and all rules with $false$ as heads function as integrity constraints. The abducible predicates are defined to be $initially\_true(F)$ and $initially\_false(F)$ to capture the incomplete knowledge about the initial situation. 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. 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 [4]. 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 4.1 *πD is an acyclic program with first-order constraints in the sense of [1].*

Corollary 4.2 *For abductive program πD, $COMP(\pi D)$ coincides with its generalized stable model semantics [7] and generalized well-founded model semantics [12].*

Theorem 4.3 (Soundness) *Let D be any domain description. For any value proposition Q, if $COMP(\pi D) \models \pi Q$, then D entails Q.*

Theorem 4.4 (Completeness) *Let D be a domain description. For any value proposition Q, if D entails Q, then $COMP(\pi D) \models \pi Q$.*

Thus, proof of entailments in $\mathcal{A}_{CO}$ is reduced to queries in logic programming. In our work we have experimented it with our belief revision system REVISE [3] which uses the WFSX semantics [13]. The use of REVISE is not essential, since the major semantics of πD coincide.

5 Observations and explanations

In what follows we study, from the standpoints of diagnosis, a class of problems related to observations and explanations, of which the Stolen Car Problem (SCP) is an instance. The problem is as follows: Given a domain description D which describes the ideal behaviour of a domain of actions and changes, and an observation O which tells some fluents hold after performance of some actions are observed, we want to infer what other actions have also happened during the course of occurrences of known actions.

5.1 Diagnostic explanation in $\mathcal{A}_{CO}$

A domain description describes how the domain should behave, while domain observation tells how the domain actually behaves. Whenever there is a discrepancy between them, a diagnosis problem arises (Fig. 1). The task of diagnosis is to find the cause for the behavioural discrepancy. For the Stolen Car Problem, the proposition *Stolen* **after** $Wait; Wait; Wait$ should not be part of the domain description. Instead it is better regarded as an observed behaviour of the domain. Hence we write it as **observed** *Stolen* **after** $Wait; Wait; Wait$.

Let $Q =$ f **after** $a_1; a_2; \ldots; a_m$ be a value proposition. Then, the following statement is called an *observation proposition*:

observed Q

where Q will simply be called an *observed value proposition*. Given the alphabet of a domain, an observation OBS of it is a finite set of observation propositions of the form (5.1). For example, for the SCP domain, the observation $OBS_{scp} =$ { **observed** *Stolen* **after** $Wait; Wait; Wait$}, while its domain description $D_{scp} =$ { **initially** $\neg$*Stolen*. *Steal* **causes** *Stolen*}.

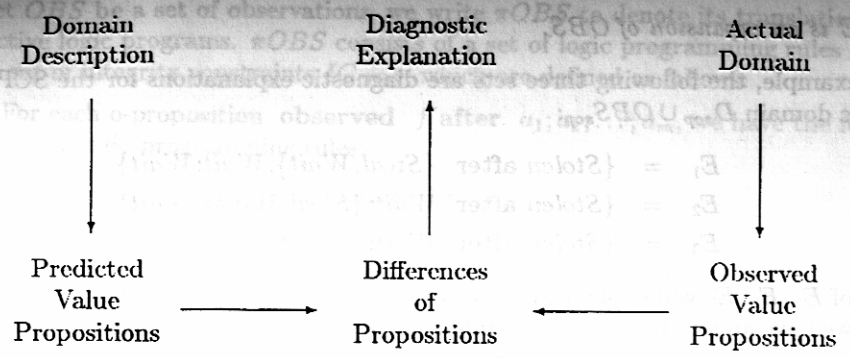


Figure 1: Diagnosis for Domains of Actions and Changes

Definition 5.1 (Diagnosis Problem) *Let D be a domain description, and $OBS =$ { **observed** $Q_1, \ldots,$ **observed** Q_m} an observation. Let $OVP = \{Q_1, \ldots, Q_m\}$. We call (D, OBS), or simply (D, OVP), a diagnostic domain. We also say that there is a diagnostic explanation problem, or simply diagnosis problem, for the domain iff $D \cup OVP$ is inconsistent.*

For example, as $D_{scp} \cup$ {*Stolen* **after** $Wait; Wait; Wait$} is inconsistent, there is a diagnostic explanation problem for SCP.

Notice that there is always a diagnostic explanation problem for the domain if the domain description D itself is inconsistent. If D is inconsistent, we may think that the domain is not well-described. In this paper, we assume a domain description is well-described. Further we will restrict ourselves to complete domain descriptions for convenience. Recall that a domain description is complete iff it has one and only one model.

When there is a diagnostic explanation problem, we need to find the cause for the inconsistency of the diagnosis problem. In the rest of the paper, we develop a methodology which is suitable for SCP-like problems. Since we assume the domain description is correct and describes the ideal behaviour of the domain, what is wrong with the difference between predicted behaviour and observed behaviour is that some unobserved actions have happened.

Definition 5.2 *Let f* **after** $a_1; \ldots; a_m$ *and f* **after** $b_1; \ldots; b_m$ *be two value propositions. The latter is an expansion of the former iff for every $1 \leq i \leq m$, $a_i \subseteq b_i$. Let P_1 and P_2 are two sets of value propositions. P_2 is said to be an expansion of P_1 iff (i) for every value proposition Q_1 of P_1 there is a value proposition Q_2 in P_2 which is an expansion of Q_1; and (ii) every value proposition Q_2' of P_2 is an expansion of a value proposition Q_1' of P_1.*

Definition 5.3 (Diagnostic explanation) *Let (D, OVP) be given. A diagnostic explanation for it is a set of value propositions E such that (i) $D \cup E$ is consistent;*

(ii) E is an expansion of OBS.

For example, the following three sets are diagnostic explanations for the SCP diagnostic domain $D_{scp} \cup OBS_{scp}$:

$$E_1 = \{Stolen \textbf{ after } \{Steal, Wait\}; Wait; Wait\}$$
$$E_2 = \{Stolen \textbf{ after } Wait; \{Steal, Wait\}; Wait\}$$
$$E_3 = \{Stolen \textbf{ after } Wait; Wait; \{Steal, Wait\}\}$$

Any of E_1, E_2, E_3 will explain why the car is missing in the parking lot. Now suppose that we have an additional action symbol, say $Drink$. Then, the following set of value explanations is also an diagnostic explanation for $D_{scp} \cup OBS_{scp}$:

$$E_4 = \{Stolen \textbf{ after } \{Steal, Wait, Drink\}; Wait; Wait\}$$

Comparing E_4 with E_1, we may prefer E_1, since it provides a *parsimonious* explanation.

Definition 5.4 (Preferred Explanation) *Suppose E_1 and E_2 are two diagnostic explanations for a given diagnostic domain (D, OVP). We say that E_1 is preferred to E_2 iff E_2 is an expansion of E_1. A diagnostic explanation E for a diagnostic domain (D, OVP) is said to be most preferred iff for any diagnostic explanation E' for (D, OVP) if E' is preferred to E, then $E = E'$.*

For example, E_1 is a preferred explanation to E_4 for SCP diagnostic domain $D_{scp} \cup OBS_{scp}$. Moreover, all of E_1, E_2 and E_3 are most preferred diagnostic explanations.

5.2 Diagnostic explanation in abductive logic programs

Now we want to compute preferred diagnostic explanations by abductive logic programs. First of all, we have to translate observation propositions into rules of abductive logic programs. Although an observation proposition consists of a value proposition, the translation is not the same, as we do not exclude the possibility of occurrences of other actions in the course of $Wait; Wait; Wait$ when we say **observed** $Stolen$ **after** $Wait; Wait; Wait$. As $Result(A, S)$ is used to denote the new situation obtained by only doing A in S, we cannot simply translate **observed** $Stolen$ **after** $Wait; Wait; Wait$ into something like $holds(Stolen, Result(Wait; Wait; Wait, s_0))$. Suppose that we know the action A is being done, we cannot use $happens(A, S)$ to denote it as in the event calculus [9]. The reason is that the underlying time structure in $\mathcal{A}_{CO}$ is branching, while in the event calculus the time structure is essentially linear. In order to keep the branching time structure, we use $occ(A, S_1, S_2)$ to denote the action A occurs in situation S_1 and leads to the situation S_2 possibly together with some other actions. For diagnostic purpose we introduce another abducible predicate $happens/3$. The atom $occ(A, S_1, S_2)$ and $happens(A, S_1, S2)$ have the same meaning and could be denoted by only one predicate. The reason that we have deliberately introduced such two predicates is that we want to make abducible predicates explicit. The predicate $happens/3$ is abducible.

Let OBS be a set of observations, we write πOBS to denote its translation into abductive logic programs. πOBS consists of a set of logic programming rules P_{OBS} and a set of integrity constraints IC_{OBS}, which are defined as follows:

1. For each o-proposition **observed** f **after** $a_1; a_2; \ldots; a_m$, we have the following m logic programming rules:

$$occ(a_1, s_0, s_{a_1}) \leftarrow$$
$$\ldots$$
$$occ(a_m, s_{a_1;a_2;\ldots;a_{m-1}}, s_{a_1;a_2;\ldots;a_m}) \leftarrow$$

   And adding the following into IC_{OBS} as a constraint:

$$false \leftarrow not\ holds(f, s_{a_1;a_2;\ldots;a_m})$$

2. We add the following additional rules for $occ/3$:

$$occ(A, S_1, S_2) \leftarrow atomic(A), happens(A, S_1, S_2)$$

   where $atomic(A)$ denotes that A is an atomic action, i.e. $A \in \sigma_a$.

Then, for a given diagnostic domain (D, OBS), its translation is defined to be $\pi D \cup \pi OBS$. For example, for the SCP domain description D_{scp} and observation O_{scp}, it can be shown that $COMP(P_{scp}) \cup IC_{scp} \cup \{happens(Steal, s_0, s_1)\}$ is consistent, and the set of abducibles $\{happens(\ Steal,\ s_0, s_1)\}$ can be obtained by executing the underlying abductive query evaluation procedure such as REVISE [3]. Actually, $\{happens(Steal, s_0, s_1)\}$ corresponds to the diagnostic explanation E_1. Note that it can be shown that $\pi D \cup \pi OBS$ is still an acyclic program with first-order constraints. Thus, we will still define its semantics as the completion of defined predicates with first-order constraints as for πD before. Thus, we can use an abductive query procedure to generate abductive answers to queries. In particular, an abductive answer, if any, may be generated for the query $\leftarrow \neg false$. If such an answer Δ exists, the theory $COMP(P_D \cup P_{OBS}) \cup IC_D \cup IC_{OBS} \cup \Delta$ is consistent.

5.3 Soundness and completeness of explanations

Our translation π is said to be sound in diagnostic explanation if for every abductive answer to $\leftarrow \neg false$ there is a diagnostic explanation, and π is said to be complete in diagnostic explanation if for every diagnostic explanation there is an abductive answer to $\leftarrow \neg false$.

Theorem 5.5 (Explanation Soundness) *Let (D, OBS) be a diagnostic domain. If there exists Δ_H such that $\pi D \cup \pi OBS \cup \Delta_H$ is consistent, then there is a diagnostic explanation E for (D, OBS).*

Theorem 5.6 (Explanation Completeness) *Let (D, OBS) be a diagnostic domain. Let E be a diagnostic explanation. Then, there is a Δ_H such that $\pi D \cup \pi OBS \cup \Delta_H$ is consistent.*

Corollary 5.7 *If $D \cup OVP$ is consistent, then there is an abductive answer Δ to $\leftarrow \neg false$ such that no atoms of the form $happens(a, s_i, s_j)$ appear in Δ.*

Acknowledgements

This work was partially supported by a post-doctoral fellowship from JNICT under PRAXIS XXI/BPD/4165/94 to the first author, and JNICT PROLOPPE project under PRAXIS/3/31/TIT/24/94.

References

[1] Apt, K.R. and Bezem, M., Acyclic programs, *Proc. of ICLP 90*, 579–597

[2] Baral, C. and Gelfond, M., Representing concurrent actions in extended logic programming, *IJCAI*, 1993, 866–871

[3] Damásio, C. V., Nejdl, W. and Pereira, L.M., REVISE: An Extended Logic Programming System for Revising Knowledge Bases, *KR'94*, 1994

[4] Denecker, M., *Knowledge Representation and Reasoning in Incomplete Logic Programming*, Department of Computer Science, K.U.Leuven, 1993

[5] Gelfond, M. and Lifschitz, V., Representing action and change by logic programs, *Journal of Logic Programming*, Vol.17, 1993, 301–322

[6] Kakas, A.C., Kowalski, R.A., Toni, F., Abductive logic programming, *J. of Logic and Computation*, 2;6, 1993, 719-770

[7] Kakas, A.C., Mancarella, P., Generalized stable models: A semantics for abduction, *Proc. of ECAI'90*, 1990

[8] Kautz, H.A., The logic of persistence, *Proc. of the AAAI86*, 1986, 401–405

[9] Kowalski, R.A. and Sadri, F., The situation calculus and event calculus compared, *Proc. of ILPS 94*, 1994, 539–553

[10] Li, R. and Pereira, L. M., Temporal Reasoning with Abductive Logic Programming, *ECAI'96*, 13 – 17

[11] Lin, F. and Y. Shoham, Concurrent actions in the situation calculus, *Proc. of AAAI-92*, 1992, 590–595

[12] Pereira, L. M., Aparício, J. N., and Alferes, J. J., Hypothetical reasoning with well-founded semantics, In: B. Mayoh (ed.), *Proc. of the 3rd Scandinavian Conference on AI*, IOS Press, 1991

[13] Pereira, L. M., Aparício, J. N., and Alferes, J. J., Non-monotonic Reasoning with Logic Programming, *Journal of Logic Programming* 17(2,3 & 4), 1993, 227–263

Executing Intensional Logic with the TAS Tool*

Manuel Enciso Juan F. Moncada Inma P. de Guzmán Manuel Ojeda

E.T.S.Ingeniería Informática, Universidad de Málaga,
P.O. Box 4114, 29080. Málaga, SPAIN. Tel/Fax: +34-5-213 3309 / 1396,
Email: {enciso,pguzman,aciego}@ccuma.sci.uma.es

Abstract

In this work we apply the TAS methodology [2, 3] to intensional logic. Particularly, we show how the TAS tool works with the minimal intensional logic K. Our goal is to sketch how the TAS paradigm for Propositional Logic and First Order Logic [10] can be extended to the K logic. Specifically, we check the efficiency of the method by comparing it to the classical tableaux method. In our opinion, the results obtained in the test allows us to consider the TAS tool as a reliable approach to automated theorem proving for non-standard logic.

Keywords Non-Standard logic. Executable Intensional Logic. Automated theorem proving.

1 Introduction

In recent years, executable forms of non-standard logic have been proposed to provide more appropriate logical techniques, resulting in the introduction of more sophisticated theorem proving techniques [9, 8, 4]. All of these techniques are either tableaux-like methods or resolution-like methods.

In this paper we show how the TAS tool can provide an ATP, named TAS-K, which in our opinion avoids some of the classic problems of tableaux methods. In particular, the main features of TAS-K are the following:

1. TAS-K is strongly based on the structure of the formula, that is, on the structure of its syntactic tree. It is a re-writing method, which works by using transformations on these syntactic trees (TAS stands for *Transformaciones de Árboles Sintácticos*, the Spanish rendering of Syntactic Tree Transformations).

2. Its power is based not only on the intrinsically parallel design of the involved transformations, but also on the fact that these transformations are not just applied one after the other. The method incorporates some criteria which can be efficiently implemented.

   These criteria allow us either to reach as a conclusion whether or not the structure of the syntactic tree has direct information about the validity of the given formula or to decrease the size of the problem before distributing.

3. The weight of the exponential complexity of this method holds only on the $(\wedge, \odot)$-$\vee$-*par* tree transformation, which executes the distributions. In order to improve its efficiency, it makes feasible the parallel execution of non-avoidable distributions and again uses the above criteria to decrease, if possible, the number of distributions.

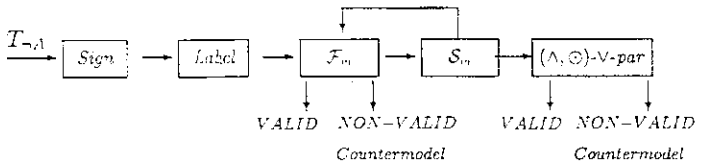
The method deals with the K logic, the basis of any other modal logic. It is a minimal logic in the sense that no assumptions on the accessibility relation are made. The alphabet of K has two modal connectives: $\Diamond$ (*it is possible that*) and $\Box$ (*it is necessary that*). We consider the standard syntax and semantics. In the rest of the paper Ω will denote the set of propositional symbols and $\top$, $\bot$ denote the boolean constants *true* and *false*.

*This work has been partially supported by CICYT project TIC94-0847-C02-02.

2 The TAS-K method

The key idea in the TAS paradigm is to use the information given by partial interpretations (exhaustively used in Quine's method) just for unitary partial interpretations, adapting it to the K logic.

Let A be a wff of K, the input of TAS-K is the syntactic tree of $\neg A$, $T_{\neg A}$. The flow diagram of TAS-K is the following



2.1 The Sign and Label tree-transformations

To *Sign* is to transform any wff of K into an equivalent *intensional negation normal form* (*innf*), a particular case of nnf for intensional logics defined by the authors.

The key tools of TAS-K are the Δ sets; whose intuitive meaning is the following:

- Δ_0 contains "sufficient conditions to assert that A *is false*" in a modal interpretation.
- Δ_1 contains "sufficient conditions to assert that A *is true*" in a modal interpretation.

The elements in the Δ-sets are *modal literals*, i.e., an innf shaped as either $M_1 M_2 \dots M_m\, p$ or $M_1 M_2 \dots M_m\, \neg p$ where each M_i is an intensional connective.

If A is an innf, then to *label* the syntactic tree of A, T_A will mean to label each node N in T_A with the ordered pair $(\Delta_0(B), \Delta_1(B))$, where B is the subformula of A determined by N in T_A.

As indicated at the beginning of this section, the purpose of defining Δ_0 and Δ_1 is to collect information about models of $\neg A$ and A, as stated in the following theorem:

Theorem 1 *If A is in innf and l is a modal literal, then*

(a) If $l \in \Delta_0(A)$, then $\models (\neg l \rightarrow \neg A)$ *(b) If $l \in \Delta_1(A)$, then $\models (l \rightarrow A)$*

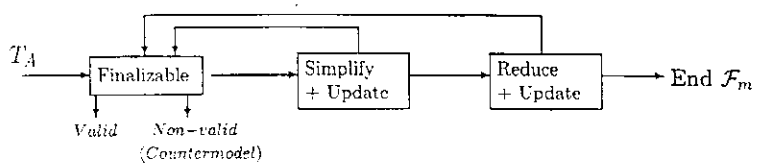
2.2 The $\mathcal{F}_m$ tree-transformation

At this stage we try to detect, by means of the Δ sets, whether the structure of the syntactic tree of A, T_A, provides either complete information about the unsatisfiability of A or useful information to decrease its size before distributing. Specifically, the labels will allow:

- To conclude that the formula A is satisfiable, checking if it is *finalizable*.
- To conclude that a subformula B, in particular the whole formula, is equivalent to $\top$, $\bot$ or a modal literal, using the process *simplify*.
- To decrease the size of the formula, substituting A by a simultaneously unsatisfiable formula in which the symbols in the Δ-labels occurs at most once, using the process *reduce*.

After *simplifying* or *reducing*, the labels of some nodes have to be recalculated (these nodes are just the ancestors of the reduced nodes), and the constants $\top$ and $\bot$ (introduced when a subformula has been detected to be valid or unsatisfiable) have to be treated. This work is made by the process *update* recursively applying the transformations induced by the 0-1 laws of propositional logic and the following equivalences of K: $\Diamond\bot \equiv \bot$ and $\Box\top \equiv \top$.

Checking the previous items using *finalizability*, *simplifiability* and *reducibility* with their corresponding *updates*, as many times as possible, is the core of the $\mathcal{F}_m$ tree-transformation.



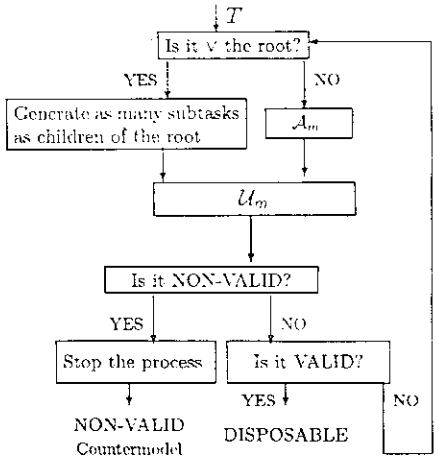
2.3 The tree-transformation $\mathcal{S}_m$

This transformation is an extension of the well-known pure literal rule. We detect those *n-positive* (or *n-negative*) propositional symbols and substitute all its occurrences of p with modal order n $\top$ (or $\bot$). We treat separately the occurrences of the propositional symbols for each modal order.

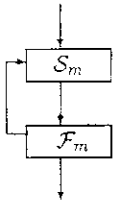
The main benefit in the use of $\mathcal{S}_m$ is that we apply this transformation more than once, specifically, it is always applied after the formula has been reduced by other transformation.

2.4 The Tree-Transformation $(\wedge,\odot)$-$\vee$-par

As indicated in the introduction, the exponential complexity of TAS-K is due to its last stage. TAS-K will be a good method provided this last stage has an acceptable behaviour. In our opinion, this will only be possible if we can make feasible the parallel execution of non-avoidable distributions and if we can avoid as many distributions as the structure of the formula allows. This is why $(\wedge,\odot)$-$\vee$-*par* makes the distributions, and then uses once again the transformation $\mathcal{F}_m$. The gradual distribution process is made by the tree-transformation $\mathcal{A}_m$. The flow diagram of this transformation is the following:



Where $\mathcal{U}_m$ is:



3 TAS-K versus Tableaux

As we have mentioned in the introduction, we have checked the efficiency of the new method TAS-K by comparison with the method of the semantic tableaux, because none of these methods require preprocessing the formulas to be proved.

We assert that the TAS tool provides more efficient ATPs than Tableaux-like methods because the TAS methodology subsumes any specific efficiency strategy for Tableaux.

We have implemented TAS-K and Semantic Tableaux in order to evaluate the efficiency of TAS-K. Using these implementations, we obtained the set of times spent by each method on a

randomly generated set of formulas. On these sets of times two exponential curves were fitted, and the conclusions of the comparison are the following:

- The time spent by TAS-K is smaller than the time spent by tableaux in almost every formula.

  More concretely, for most formulas, TAS-K spends a linear time. This fact reflects that the distribution in the $(\wedge, \odot)$-$\vee$-*par* stage has not been applied to this set of formulas. This was one of the aims of the TAS-K design, as we mentioned in the introduction.

- The fitted curve obtained by TAS-K is always below the one for tableaux. This result, in our opinion, is sufficient to say that TAS-K improve the efficiency of tableaux method.

References

[1] M. Abadi and Z. Manna. Modal theorem proving. In Springer-Verlag, editor. *8th Int. Conf. on Automated Deduction*. Lecture Notes in Computer Science, 1986.

[2] G. Aguilera, I.P. de Guzmán, and M. Ojeda. TAS-D^{++} syntactic trees transformations for automated theorem proving. *Lect. Notes in Artif. Intelligence no 838*, pages 198–216, 1994.

[3] G. Aguilera, I.P. de Guzmán, and M. Ojeda. A New Tool for a General Approach to ATPs. *KI-95, Workshop on Computational Propositional Logic*, Bielefeld, Germany 1995.

[4] A. Artosi and G. Governatori. Labelled model modal logic. In *CADE12 Workshop on Automated Model Building*. Springer-Verlag. LNAI, 1994. 838

[5] M. Fitting. Destructive modal resolution. *Journal of Logic and Computation*, 1(1), 1990.

[6] I.P. de Guzmán and M. Enciso. A new and complete theorem prover for temporal logic. In *Proceedings of the IJCAI Workshop on Executable Temporal Logics*, Montreal (Canada), 1995.

[7] M. Enciso. *Lógica temporal y demostración automática de teoremas. Eficiencia y paralelismo*. PhD thesis, Univ. de Málaga, 1995.

[8] G. Governatori. Labelled tableaux for multimodal logics. In *4th Workshop on Theorem Proving. Analytic Tableaux and Related Methods*, Berlin, German, 1995. Springer-Verlag. LNAI. 918

[9] R. Hähnle and O. Ibens. Improving temporal logic tableaux using integer constraints. In *1st International Conference on Temporal Logic*, Munich, Germany LNAI 827, 1994.

[10] M. Ojeda. *Métodos formales para deducción en Lógica de Primer Orden*. PhD thesis, Univ. de Málaga, 1996.