REPRESENTING ACTIONS: LAWS, OBSERVATIONS AND HYPOTHESES

CHITTA BARAL, MICHAEL GELFOND AND ALESSANDRO PROVETTI

▷ We propose a modification $\mathcal{L}_1$ of the action description language $\mathcal{A}$. The language $\mathcal{L}_1$ allows representation of hypothetical situations and hypothetical occurrence of actions (as in $\mathcal{A}$) as well as representation of actual occurrences of actions and observations of the truth values of fluents in actual situations. The corresponding entailment relation formalizes various types of common-sense reasoning about actions and their effects not modeled by previous approaches. As an application of $\mathcal{L}_1$ we also present an architecture for intelligent agents capable of observing, planning and acting in a changing environment based on the entailment relation of $\mathcal{L}_1$ and use logic programming approximation of this entailment to implement a planning module for this architecture. We prove the soundness of our implementation and give a sufficient condition for its completeness. ◁

1. INTRODUCTION

To perform nontrivial reasoning an intelligent agent situated in a changing domain needs the knowledge of causal laws that describe effects of actions that change the domain, and the ability to observe and record occurrences of these actions and the truth values of fluents (by fluents in this paper we mean propositions whose truth values depend on time) at particular moments of time. One of the central problems of knowledge representation is the discovery of methods of representing this kind of

*Address correspondence to*Department of Computer Science, The University of Texas at El Paso. El Paso, Texas 79968 USA.
E-mail: {chitta,mgelfond,provetti}@cs.utep.edu
Web: http://cs.utep.edu/csdept/krgroup.html

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

information in a form allowing various types of reasoning about the dynamic world and at the same time tolerant to future updates.

There are numerous attempts at finding solutions to this problem (for recent developments see for instance the special issue of the Journal of Logic and Computation [Geo94] and the workshop proceedings [Wor95]). These attempts are primarily concerned with finding mathematical models describing effects of actions, designing languages and entailment relations suitable for axiomatization of these models and discovering inference mechanisms capable of (efficiently) drawing inferences from these axioms. Most of the recent efforts seem to be directed at the development of methodologies for building provably correct systems capable of reasoning about actions.

In this paper we continue the work started in [GL92] where the authors introduced the high-level *action description language* $\mathcal{A}$ capable of expressing causal laws which describe effects of actions as well as statements about values of fluents in possible states of the world. The entailment relation of the language models hypothetical reasoning similar to that of situation calculus [McC63, MH69]. In the last few years the syntax and semantics of $\mathcal{A}$ were expanded to allow descriptions of the effects of concurrent and non-deterministic actions as well as descriptions of global constraints expressing time-independent relations between fluents [BG93, KL94, BT94, Bar95, GL95, BT94, HT93, MT95].

This work can be viewed as complementary to the alternative approach based on direct axiomatizations of theories of actions in classical logic and its nonmonotonic extensions [PR93, Pin94, Rei91, LS91, MS94, Pro96]. We believe that both approaches should be developed further before serious comparison between them could become possible. However we briefly mention several attractive features shared by action description languages. They have simple and restrictive syntax which gives an advantage to the specifier similar to the advantage in using Pascal over PL/1 or RISC languages over the assembly language of IBM 360. Their semantics is based on the notion of automaton, which provides additional insight into the behavior of the corresponding dynamic system. Finally, most of these languages do not commit to any specific set of logical connectives which we believe is advantageous in certain situations (see, for instance, [MT95, Bar95]).

In this paper we improve $\mathcal{A}$ by expanding its ontology with actual situations interpreted as sequences of actions describing the actual evolution of the system (as opposed to the hypothetical situations of $\mathcal{A}$). Consequently the new language $\mathcal{L}_1$ is capable of expressing actual situations and their temporal ordering, and observations of both the truth values of the fluents and the actual occurrences of actions in these situations. The corresponding entailment relation of $\mathcal{L}_1$ formalizes various types of common-sense reasoning about actions and their effects not modeled by previous approaches.

To clarify our motivation let us consider the following simple example[1].

Suppose that we are given the following story[2]:

> John needs to bring his *packed* suitcase *to the airport*. John knows that if he *has-a-car* then by doing the action *drive-to-the-airport* he will be *at-the-airport* and not *at-his-home*. Similarly, if the action

[1] This example is in the spirit of the Glasgow-London-Moscow problem [McC].

[2] The italicized terms in this story correspond to actions and fluents and are used in Example 1.

> *hit-car* occurs then he will not *have-a-car*, if the action *rent-a-car* occurs then he will *have-a-car*, and if the action *pack* occurs when he is *at-his-home* he will have his suitcase *packed*. John also knows that he is *at-his-home*, and therefore not *at-the-airport* and he *has a car*. The plan of *packing* the suitcase and *driving to the airport* is adequate to achieve John's goal. He then follows his plan and starts *packing* his suitcase. But right after *packing* he observes his *car being hit*. Following the rest of his original plan, he will no longer achieve his goal. Instead the plan of first performing *rent-a-car* and then performing *drive-to-the-airport* would be adequate.

Our longstanding goal is to learn how to design and implement programs capable of simulating the types of behavior exhibited by John in the above story. As a first step we would like to have a mathematical theory to help us to write this and similar programs. We believe that the language $\mathcal{L}_1$ and the corresponding entailment relation form an important part of such a theory. They allow us to precisely describe effects of actions available to John as well as the results of John's observations, and to characterize the set of valid conclusions about the past, current and future states of the world which can reasonably be made by John on the basis of his knowledge.

This is of course not enough. To succeed in designing a program simulating John's behavior, we need to better understand its dynamics. In particular, a good theory should suggest a program architecture —a possible way of decomposing the problem into simpler independent modules responsible for different intellectual tasks such as planning from the current situation and updating the current domain description by new observations. Finally, we should develop a methodology of implementing these modules according to the specification.

The first six sections of this paper are devoted to description of the syntax and semantics of $\mathcal{L}_1$ and its entailment relation, as well as an application of this entailment to the design of an architecture for intelligent agents capable of observing, planning and acting in a changing environment. We start with the description of a language $\mathcal{L}_0$ capable of expressing general laws governing effects of actions together with observations of values of fluents and occurrences of actions in particular situations. In Section 5 we extend $\mathcal{L}_0$ to $\mathcal{L}_1$ by allowing hypothetical reasoning.

It is probably worth noting that $\mathcal{L}_1$ is not an extension of $\mathcal{A}$. We believe that allowing hypothetical statements of the form "fluent f would be true after the execution of a sequence α of actions" in domain descriptions of $\mathcal{A}$ is unnecessary and (in the presence of actual situations and facts) leads to certain complications (see Section 5 and Appendix A for a discussion on this aspect). In $\mathcal{L}_1$ we limit the use of hypothetical statements to queries. Domain descriptions of $\mathcal{L}_1$ can only contain causal laws describing effects of actions, and facts but not hypotheses. These restrictions lead to clearer ontology and semantics of the language.

Section 7 addresses the question of computing the entailment relation of $\mathcal{L}_1$ w.r.t. domain descriptions generated according to our architecture. First, we follow the basic idea of [GL92] and translate such domain descriptions into (declarative) logic programs. Some sufficient conditions on the type of a domain description D guarantee soundness (and sometimes completeness) of the translation Π_D (this adds to the list of translations from domain descriptions of action description languages with different ontologies into disjunctive, abductive and equational logic programs

[Dun93, DDS93, HT93, Tur94, ALP94]).

To be able to actually answer queries to Π_D we need to use a particular query answering algorithm. The standard Prolog interpreter (as implemented – for instance – in Quintus Prolog) is certainly the most popular among the large family of such algorithms [CSW95, ADP94] and seems to be a natural choice for use in this paper. Consequently, we view Π_D as a Prolog program and prove soundness and completeness of the Prolog inference mechanism w.r.t. the answer set semantics[3] [GL91] of Π_D.

In the last part of the paper, we discuss an implementation of the planning module of the proposed system. This module consists of a procedural part that generates candidate plans and a declarative description of the domain description D, represented by its translation Π_D, which is used for testing. The planner is simple, has a clear declarative specification and can be proven correct w.r.t. this specification. The planner can be used in changing domains. It allows (on-line) introduction of new objects in the domain (such as the appearing of a new block at the table in the blocks world), as well as addition of new information about values of fluents and occurrences of actions. It is nonmonotonic, i.e., capable of creating new plans when new observations invalidate the old plans. Even though there are reports in the literature about planners combining one or more of these features [Wel94, KKY95, Ped88] we are not aware of a system which contains all of them. Moreover, inclusion of these features in our framework does not require any special effort. Even though the main purpose of building the planner was clarity and provable correctness, its efficiency is not as bad as we expected. For the domains from [Wel94] that we tested, our planner performed better for the particular queries and initial states that we considered [4].

We believe that developing action specification languages and approximating their entailment relation in logic programming contributes to better understanding of the underlying ontological principles of reasoning about actions as well as the advantages and limitations of logic programming languages as tools for implementing such reasoning. It already allowed to establish equivalence of some of the previously known theories of actions seemingly based on different intuitions, languages and logics [Kar93, Kar94] and stimulated work on the theory and implementation of logic programming languages [AB91, Tur93, MT94, LT94, LMT93].

From the software engineering standpoint, theories of actions can be interesting as a testbed for building provably-correct classes of logic programs from rather abstract specifications. Our work contributes to better understanding of this process.

Finally, we hope that the development of planners based on general theories of actions will contribute to establishing closer ties between theories of actions, planning, and logic programming.

[3] The use of answer set semantics is not essential. For Π_D it coincides with the well-founded [VGRS91] and some of the other semantics. To make the paper self-contained, definition of the answer set semantics is given in Appendix B.

[4] We are currently in the process of doing extensive testing by randomly generating the queries and initial states.

2. SYNTAX OF $\mathcal{L}_0$

We will start with the description of a language $\mathcal{L}_0$ capable of expressing general laws governing effects of actions together with observations of values of fluents and occurrences of actions in particular situations. In Section 5 we extend $\mathcal{L}_0$ to $\mathcal{L}_1$ to allow hypothetical reasoning.

The alphabet of $\mathcal{L}_0$ consists of three disjoint nonempty sets of symbols $\mathcal{F}$, $\mathcal{A}$ and $\mathcal{S}$, called *fluents*, *actions*, and *actual situations*. Elements of $\mathcal{A}$ and $\mathcal{S}$ will be denoted by (possibly indexed) letters a and s respectively. We will also assume that $\mathcal{S}$ contains two special situations s_0 and s_N called *initial* and *current* situations.

A *fluent literal* is a fluent possibly preceded by negation $\neg$. Fluent literals will be denoted by (possibly indexed) letters f and p (possibly preceded by $\neg$). $\neg\neg f$ will be equated with f. For any fluent f, $\overline{\neg f} = f$, and $\overline{f} = \neg f$.

We denote sequences [5] of actions by the Greek letter α and its indexed versions. For an action a, $\alpha \circ a$, means the sequence of actions where a follows α. We also sometimes denote sequence of actions with the standard Prolog notation for lists. For example, the list of actions $[a_1, a_2, \ldots, a_n]$, denotes the sequence of actions where a_1 is followed by a_2 and so on up to a_n.

There are two kinds of propositions in $\mathcal{L}_0$ called causal (or effect) laws and facts:

- A *causal law* is an expression of the form

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \tag{1}$$

  where a is an action, and $f, p_1, \ldots, p_n$ ($n \geq 0$) are fluent literals. $p_1, \ldots, p_n$ are called *preconditions* of (1). We will read this law as "f is guaranteed to be true after the execution of an action a in any state of the world in which $p_1 \ldots p_n$ are true." [6]

  If $n = 0$, we write the effect law simply as "a **causes** f." Two causal laws of the form a **causes** f **if** $p_1, \ldots, p_n$ and a **causes** $\neg f$ **if** $q_1, \ldots, q_m$ are said to be *contradictory* if $\{p_1, \ldots, p_n\} \cap \{\overline{q_1}, \ldots, \overline{q_m}\} = \emptyset$.
- An atomic *fluent fact* is an expression of the form

$$f \textbf{ at } s \tag{2}$$

  where f is a fluent literal and s is a situation. The intuitive reading of (2) is "f is observed to be true in situation s."
- An atomic *occurrence fact* is an expression of the form

$$\alpha \textbf{ occurs\_at } s \tag{3}$$

  where α is a sequence of actions, and s is a situation. It states that "the sequence α of actions was observed to have occurred in situation s" (we assume that actions in the sequence follow each other immediately).

[5] In this paper by sequence we always mean a finite sequence.

[6] The use of the word **causes** may be somewhat misleading, since f may have been true even before the action a was executed. To justify the name we probably should require $\overline{f}$ to be included in the list of preconditions of (1). To avoid notational confusion with $\mathcal{A}$ we prefer, however, to keep the name.

- An atomic *precedence fact* is an expression of the form

$$s_1 \textbf{ precedes } s_2 \tag{4}$$

where s_1 and s_2 are situations. It states that situation s_2 occurred after situation s_1.

Propositions of the type (1) express general knowledge about effects of actions and hence are referred to as *laws*. Propositions (2), (3) and (4) are called *atomic facts* or *observations*. A *fact* is a propositional combination of atomic facts.

A collection of laws and facts is called a *domain description* of $\mathcal{L}_0$. The sets of laws and facts of a domain description D will be denoted by D_l and D_f respectively. We will only consider domain descriptions whose propositions do not mention the situation constant s_N.

To see how the domain descriptions of $\mathcal{L}_0$ can be used to represent knowledge about actions let us consider the following example:

Example 2.1. Suppose that we are given a series of observations about Fred:

a) When the water pistol was squirted Fred was seen to be *alive* and *dry* and
b) in a later moment a shot was fired at Fred.

Suppose also that it is generally known that

c) *squirting* makes Fred *wet* and
d) *shooting* makes Fred *dead*.

The above information can be represented by a domain description D_1 consisting of the following propositions:

$$D_1 = \begin{cases} Facts: & \\ (p_1) & alive \textbf{ at } s_0 \\ (p_2) & dry \textbf{ at } s_0 \\ (p_3) & squirt \textbf{ occurs\_at } s_0 \\ (p_4) & s_0 \textbf{ precedes } s_1 \\ (p_5) & shoot \textbf{ occurs\_at } s_1 \\ Laws: & \\ (p_6) & squirt \textbf{ causes } \neg dry \\ (p_7) & shoot \textbf{ causes } \neg alive \end{cases}$$

To complete the description of D_1 we need to define its language. For simplicity we assume that this language contains only the fluents, actions and actual situations mentioned in the propositions of D_1 together with the situation s_N. Unless stated otherwise, the same assumption will be made in other examples throughout this paper.

Domain descriptions in $\mathcal{L}_0$ are used in conjunction with the following informal assumptions which clarify the description's meaning:

a) changes in the values of fluents can only be caused by execution of actions;
b) there are no actions except those from the language of the domain description;

c) there are no effects of actions except those specified by the causal laws of the domain;
d) no actions occur except those needed to explain the facts in the domain description; and
e) actions do not overlap or happen simultaneously.

These assumptions give a reasonably good intuitive understanding of domain descriptions of $\mathcal{L}_0$.

Consider for instance domain description D_1 from Example 1. It is easy to see that D_1 together with assumption (d) implies that *squirt* is the only action which occurs between s_0 and s_1 and that *shoot* is the only action which occurs between s_1 and s_N. Using D_1 with the assumptions (a)—(e) we can conclude that at moment s_1 Fred is *wet* but *alive* while at moment s_N, i.e., at the end of the story, he is *wet* and *dead*.

Our goal in this paper is to build a mathematical model which will help us to better understand and eventually mechanize these types of arguments. As the first step we suggest a semantics of domain descriptions of $\mathcal{L}_0$ which precisely specifies the sets of acceptable conclusions which can be reached from such descriptions and assumptions (a)—(e). Domain descriptions under this semantics are also intended for use by system designers as a formal specification language.

3. SEMANTICS OF $\mathcal{L}_0$

In this section we introduce a semantics of our action description language $\mathcal{L}_0$. We assume that states of the world are determined by sets of fluents and that actions executed in a particular state can add and remove such fluents according to the causal laws. The set of all possible behaviors of a dynamic system, satisfying causal laws of a given domain description D, can then be described by a transition diagram with states labeled by sets of fluents and transitions labeled by actions. To interpret the facts from D we need to select the initial situation together with a path in the diagram describing the actual behavior of the system. In the following definition the idea is formalized in a somewhat non-standard way. Instead of defining a transition function and an initial state, we describe the automaton by a partial function from action sequences to states. We prefer this slightly more complex definition because of its appropriateness for expanded domains that may contain non-deterministic actions [KL94], triggers, temporal facts, etc. To make the idea precise, we need to introduce some terminology.

Let a *state* be a set of fluents. A *causal interpretation* is a partial function Ψ from sequences of actions to states such that:

1. The empty sequence $[\,]$ belongs to the domain of Ψ and
2. Ψ is prefix-closed[7].

$\Psi([\,])$ is called the initial state of Ψ. The partial function Ψ serves as an interpretation of the causal laws of D.

[7]By "prefix closed" we mean that for any sequence of actions α and action a, if $\alpha \circ a$ is in the domain of Ψ then so is α.

Given a fluent f and a state σ, we say that f *holds* in σ (f is *true* in σ) if $f \in \sigma$; $\neg f$ *holds* in σ (f is *false* in σ) if $f \notin \sigma$. The truth of a propositional combination of fluents with respect to σ is defined as usual.

Now we define effects of actions determined by causal laws of a domain description D and our informal assumptions. A fluent literal f is an (immediate) *effect* of (executing) a in σ if there is an effect law a **causes** f **if** $p_1, \ldots, p_n$ in D whose preconditions $p_1, \ldots, p_n$ hold in σ. Let us define the following sets of fluents.

$E_a^+(\sigma) = \{f : f \in \mathcal{F}$ and f is an effect of a in $\sigma\}$;

$E_a^-(\sigma) = \{f : f \in \mathcal{F}$ and $\neg f$ is an effect of a in $\sigma\}$ and

$Res(a, \sigma) = (\sigma \cup E_a^+(\sigma)) \setminus E_a^-(\sigma)$.
where Res is referred to as the *transition function*. Definition 1 below captures the meaning of causal laws of D.

Definition 3.1. A causal interpretation Ψ *satisfies* the causal laws of D if for any sequence $\alpha \circ a$ from the language of D,

$$\Psi(\alpha \circ A) = \begin{cases} Res(A, \Psi(\alpha)) & \text{if } E_A^+(\Psi(\alpha)) \cap E_A^-(\Psi(\alpha)) = \emptyset \\ undefined & \text{otherwise} \end{cases}$$

We say Ψ is a *causal model* of D if it satisfies the causal laws of D.

It is easy to see that causal models of domain descriptions are uniquely determined by their initial values, i.e., for any causal models Ψ_1 and Ψ_2 of a domain description D, if $\Psi_1([\,]) = \Psi_2([\,])$ then $\Psi_1 = \Psi_2$.
Let us illustrate the notion of causal model by the following example.

Example 3.1. Consider domain description D_1 from Example 1. The transition diagram in Figure 1 corresponds to causal laws of D_1. It is easy to check that function Res defined by D_1 is the transition function of this diagram and that a function Ψ is a causal model of D_1 iff

$\Psi([\,]) = \sigma$

for some state σ of the diagram and

$\Psi(\alpha \circ a) = Res(a, \Psi(\alpha))$

for any sequence α of actions and any action a from D_1.

Example 3.2. Consider a domain description consisting of the following propositions:
a **causes** f
a **causes** $\neg f$

It is easy to see that for any causal model Ψ of this domain description, $\Psi(\alpha)$ is undefined for any nonempty α, since for every σ, $E_a^+(\sigma) = E_a^-(\sigma) = \{f\}$.

To continue our definition of the semantics of $\mathcal{L}_0$, consider an arbitrary domain description D and let Ψ be a causal model of D. To interpret the observations of

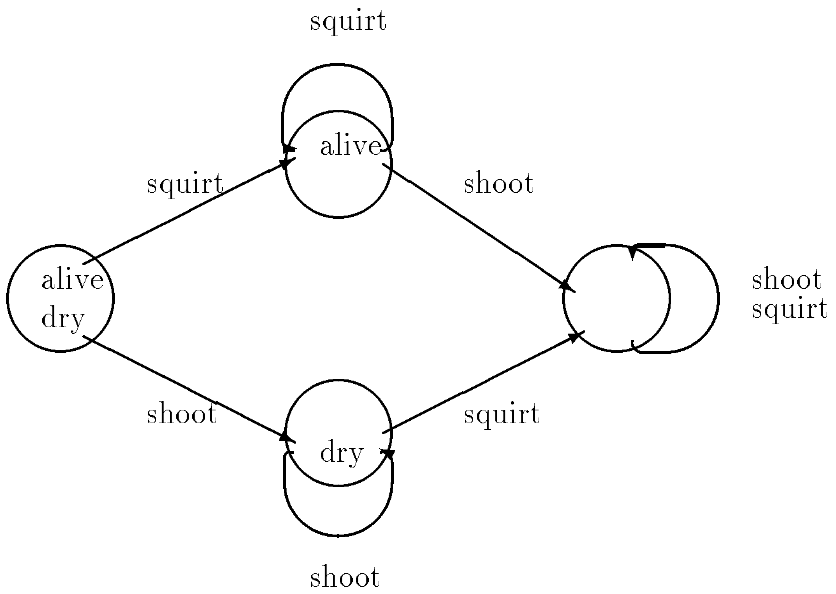

Figure 1. The states through which D_1 can evolve.

D we first need to define the meaning of situations in $\mathcal{S}$. To do so, we consider a mapping Σ from $\mathcal{S}$ to sequences of actions from the language of D. This mapping will be called *a situation assignment of* $\mathcal{D}$ if it satisfies the following properties.

1. $\Sigma(s_0) = [\,]$;
2. for every $s \in \mathcal{S}$, $\Sigma(s)$ is a prefix of $\Sigma(s_N)$.

Definition 3.2. An *interpretation* M of $\mathcal{L}_0$ is a pair (Ψ, Σ), where Ψ is a causal model of D, Σ is a situation assignment of $\mathcal{S}$ and $\Sigma(s_N)$ belongs to the domain of Ψ.

$\Sigma(s_N)$ will be called the *actual path* of M and for simplicity will often be denoted by Σ_N. Now, we can define truth of facts of D w.r.t. an interpretation M. Facts which are not true in M will be called *false* in M.

Definition 3.3. For any interpretation $M = (\Psi, \Sigma)$

1. (f **at** s) is true in M (or satisfied by M) if f is true in $\Psi(\Sigma(s))$;
2. (α **occurs_at** s) is true in M if the sequence $\Sigma(s) \circ \alpha$ is a prefix of the actual path Σ_N of M;
3. (s_1 **precedes** s_2) is true in M if $\Sigma(s_1)$ is a proper prefix of $\Sigma(s_2)$;
4. truth of non-atomic facts in M is defined as usual.

A set of facts is true in an interpretation M if all its members are true in M.

To complete the definition of model we need only to formalize the following underlying assumption of domain descriptions of $\mathcal{L}_0$:

d) No actions occur except those needed to explain the facts in the domain description.

This is done by imposing a minimality condition on the situation assignments of $\mathcal{S}$ which leads to the following definition.

Definition 3.4. An interpretation $M = (\Psi, \Sigma)$ will be called a *model* of a domain description D in $\mathcal{L}_0$ if the following conditions are satisfied:

1. Ψ is a causal model of D;
2. facts of D are true in M and
3. there is no other interpretation $M' = (\Psi, \Sigma')$ such that M' satisfies conditions 1) and 2) above and Σ'_N is a subsequence[8] of Σ_N.

A domain description D is said to be *consistent* if it has a model.

Notice that to prove that D is consistent it suffices to find an interpretation M of D satisfying the first two conditions of Definition 4. Existence of the model will follow from the finiteness of Σ_N.

In Definition 4, while minimizing the occurrences of actions we were comparing models with same initial state. This is reflected in condition 3. above. An alternative definition may be obtained by replacing condition 3. by the following:

$3'$ There is no other interpretation $M' = (\Psi', \Sigma')$ such that M' satisfies the conditions 1. and 2. and Σ'_N is a subsequence of Σ_N.

If the second definition is used, minimization of occurrences of actions is done regardless of the possible initial situations. In this paper we prefer the first definition as we would like to give equal preference to all the possible initial situations, and minimize occurrences of actions only locally with respect to particular initial situations.

The *query language* associated with $\mathcal{L}_0$ will consist of all $\mathcal{L}_0$ facts, and be denoted by $\mathcal{L}_0^Q$. The following definition describes the set of acceptable conclusions one can obtain from a domain description D.

Definition 3.5. A domain description D *entails* a query q (written as $D \models q$) iff q is true in all models of D. The set of all facts entailed by D will be denoted by $Cn(D)$.

We will say that the answer given by D to a query q is

yes if $D \models q$;
no if $D \models \neg q$;
unknown otherwise.

The next section is devoted to an analysis of $\mathcal{L}_0$ formalizations of several examples of reasoning about actions from the literature.

[8] Given a sequence $X = x_1, \ldots, x_m$, another sequence $Z = z_1, \ldots, z_n$ is a *subsequence* of X if there exists a strictly increasing sequence $i_1, \ldots, i_n$ of indices of X such that for all j = 1,2 ..., n, we have $x_{i_j} = z_j$.

4. EXAMPLES

In this section we illustrate by way of examples how domain descriptions are used to represent information and how the above notion of entailment captures informal arguments based on the information from these descriptions and the informal assumptions (a)—(e).

Let us use domain description D_1 from Example 1 to demonstrate how the entailment relation of $\mathcal{L}_0$ can model simple temporal projection.

Proposition 4.1. Domain description D_1 has a unique model and[9]

$D_1 \models ((\neg dry \wedge \neg alive)$ **at** $s_N)$

$D_1 \models ((\neg dry \wedge alive)$ **at** $s_1)$.

PROOF. . Consider the causal interpretation Ψ:

$$\Psi([\,]) = \{alive, dry\}$$

$$\Psi(\alpha \circ squirt) = \begin{cases} \{alive\} & \text{if } alive \in \Psi(\alpha) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Psi(\alpha \circ shoot) = \begin{cases} \{dry\} & \text{if } dry \in \Psi(\alpha) \\ \emptyset & \text{otherwise} \end{cases}$$

$\Sigma(s_0) = [\,]$

$\Sigma(s_1) = [squirt]$

$\Sigma(s_N) = [squirt, shoot]$.

It is easy to check that the interpretation $M = (\Psi, \Sigma)$ satisfies Definition 4 and hence is a model of D_1.

Let us show that M is the only model, i.e., for any model $M_1 = (\Psi_1, \Sigma_1)$ of D_1, $\Psi = \Psi_1$, and $\Sigma = \Sigma_1$. Since by definition any situation assignment maps s_0 into $[\,]$, statements (p1) and (p2) of D_1 will be satisfied only if $\Psi([\,]) = \Psi_1([\,]) = \{alive, dry\}$. Since Ψ and Ψ_1 are causal models of D_1 and causal models are uniquely determined by their initial values, we have that $\Psi = \Psi_1$.

To establish that $\Sigma = \Sigma_1$, notice that to satisfy the conditions (p_3)—-(p_5) the actual path $\Sigma_1(s_N)$ of M_1 must start with *squirt* followed (not necessarily immediately) by *shoot*. But then $\Sigma(s_N)$ is a subsequence of $\Sigma_1(s_N)$. M_1 is a model of D_1 and hence Σ_1 satisfies the minimality condition 3 from Definition 4. This implies that $\Sigma(s_N) = \Sigma_1(s_N) = [squirt, shoot]$. Now recall that, by definition of situation assignment, $\Sigma_1(s_1)$ must be a prefix of *[squirt,shoot]*. Since (p_3) and (p_5) are true in M_1 we have that $\Sigma(s_1) = \Sigma_1(s_1) = [squirt]$.

To complete the proof it suffices to notice that $((\neg dry \wedge \neg alive)$ **at** $s_N)$ and $((\neg dry \wedge alive)$ **at** $s_1)$ are true in M.

[9] In this paper we will use $f_1 \wedge \ldots \wedge f_n$ **at** S as an abbreviation for $(f_1$ **at** $S) \wedge \ldots \wedge (f_n$ **at** $S)$.

Example 4.1. [Reasoning by cases] Let us consider a modification of Example 1 where there is a precondition of being loaded for the shoot action to be deadly and where there are two guns at least one of which is initially loaded.

$$D_2 = \left\{ \begin{array}{ll} \multicolumn{2}{l}{Facts:} \\ (q_1) & alive\ \mathbf{at}\ s_0 \\ (q_2) & loaded_1\ \mathbf{at}\ s_0 \vee loaded_2\ \mathbf{at}\ s_0 \\ (q_3) & [shoot_1, shoot_2]\ \mathbf{occurs\_at}\ s_0 \\ \multicolumn{2}{l}{Laws:} \\ (q_4) & shoot_1\ \mathbf{causes}\ \neg alive\ \mathbf{if}\ loaded_1 \\ (q_5) & shoot_2\ \mathbf{causes}\ \neg alive\ \mathbf{if}\ loaded_2 \end{array} \right.$$

Proposition 4.2. Domain description D_2 is consistent and

$D_2 \models \neg alive$ **at** s_N.

Proof. Let $M = (\Psi, \Sigma)$ be a model of D_2. Then, by definition of causal model, we have that Ψ satisfies the following conditions (*)

$$\Psi(\alpha \circ shoot_1) = \left\{ \begin{array}{cc} \Psi(\alpha) \setminus \{alive\} & \text{if } loaded_1 \in \Psi(\alpha) \\ \Psi(\alpha) & \text{otherwise} \end{array} \right.$$

$$\Psi(\alpha \circ shoot_2) = \left\{ \begin{array}{cc} \Psi(\alpha) \setminus \{alive\} & \text{if } loaded_2 \in \Psi(\alpha) \\ \Psi(\alpha) & \text{otherwise} \end{array} \right.$$

Since M is a model, $\Psi(\Sigma(s_0))$ must satisfy facts q_1 and q_2, i.e.

$$\Psi([\,]) = \{alive, loaded_1\}$$

or

$$\Psi([\,]) = \{alive, loaded_2\}$$

or

$$\Psi([\,]) = \{alive, loaded_1, loaded_2\}$$

To satisfy q_3, $\Sigma(N)$ must start with $[shoot_1, shoot_2]$ and hence, from the minimality condition, we have that

$\Sigma(s_N) = [shoot_1, shoot_2]$

By examining each possible initial situation it is easy to check that $\Psi(\Sigma(s_N))$ satisfies $\neg alive$ **at** s_N.

Similar argument shows that (Ψ_0, Σ_0) where Ψ_0 is defined by initial condition $\Psi([\,]) = \{alive, loaded_1\}$ and by condition (*), and $\Sigma(s_0) = []$ and $\Sigma(s_N) = [shoot_1, shoot_2]$ satisfies conditions 1. and 2. of Definition 4 and hence D_2 is consistent. $\Box$

Example 4.2. [Explaining observations] Let us now consider a modified version of Example 1 where instead of

(b) "In a later moment a shot was fired at Fred"

we have

(b') "In a later moment Fred was observed to be dead."

The new scenario can be represented by a domain description D_3.

$$D_3 = \begin{cases} Facts: \\ (p_1) \quad alive \textbf{ at } s_0 \\ (p_2) \quad dry \textbf{ at } s_0 \\ (p_3) \quad squirt \textbf{ occurs\_at } s_0 \\ (p_4) \quad s_0 \textbf{ precedes } s_1 \\ (p_5) \quad \neg alive \textbf{ at } s_1 \\ Laws: \\ (p_6) \quad squirt \textbf{ causes } \neg dry \\ (p_7) \quad shoot \textbf{ causes } \neg alive \end{cases}$$

Using arguments similar to those above we can prove the following proposition.

Proposition 4.3. Domain description D_3 is consistent and

$D_3 \models [squirt, shoot]$ **occurs_at** s_0. □

5. FROM DOMAIN DESCRIPTIONS TO HYPOTHETICAL REASONING

Even though domain descriptions in $\mathcal{L}_0$ can express types of knowledge and reasoning not easily expressible in other variants of $\mathcal{A}$, they lack the ability of the latter to do hypothetical reasoning. Even the simple original version of $\mathcal{A}$ allows propositions of the form

$$f \textbf{ after } a_1, \ldots, a_m. \tag{1}$$

In $\mathcal{A}$ such statements can be understood in several different ways. Sometimes they are viewed as observations and read as "f is true after the execution of the sequence of actions $[a_1, \ldots, a_m]$." We believe that such a reading is misleading. Indeed, consider a domain description D containing statements f **after** a_1 and g **after** a_2. Since the ontology of $\mathcal{A}$ does not allow concurrent actions it is impossible to observe both a_1 and a_2 occurring at s_0. Therefore if the above statements are viewed as facts, the domain description D should be characterized as inconsistent, which of course is not the case in $\mathcal{A}$. It is probably better to understand (1) as saying that $a_1, \ldots, a_m$ can be executed in the initial situation, and if it were, then f would be true afterwards. This reading allows the use of such statements for querying domain descriptions about possible outcomes of actions.

If we want to allow statements of the form (1) in domain descriptions of our language we should understand them as hypothetical or even counterfactual. Since we want to find a language containing only those constructs which are absolutely necessary for reasoning about actions and their effects and since we want to preserve the reading of domain descriptions as theories containing only actual knowledge about the domain, *we limit ourselves to using hypotheses in queries but not in the domain descriptions.* However, in Appendix 9 we will discuss how this restriction could be lifted.

Meanwhile, let us go to precise definitions, and start by introducing propositions of the form

$$f \textbf{ after } [a_1, \ldots, a_n] \textbf{ at } s \tag{2}$$

called *hypotheses*, which slightly generalizes the syntax of (1). They are read as "sequence $a_1, \ldots, a_m$ of actions can be executed in the situation s, and if it were, then fluent literal f would be true afterwards."
If s in (2) is the current situation, i.e., s_N, then we simply write

$$f \textbf{ after } [a_1, \ldots, a_n] \tag{3}$$

If n in (3) is 0, then we write

$$\textbf{currently } f \tag{4}$$

The language $\mathcal{L}_1$ is obtained from $\mathcal{L}_0$ by allowing statements of the forms (2), (3) and (4). By definition, domain descriptions of $\mathcal{L}_1$ coincide with those of $\mathcal{L}_0$ and its queries are propositional formulas constructed from atomic facts of $\mathcal{L}_0$ and hypotheses. Observe that —according to these definitions— the query language of $\mathcal{L}_1$ is strictly stronger than that of $\mathcal{A}$ while, as a means of specifying domain descriptions, neither $\mathcal{L}_1$ nor $\mathcal{A}$ is more powerful.

Now we will define the semantics of $\mathcal{L}_1$. The notions of model of a domain description of $\mathcal{L}_1$ and of the truth of atomic facts in such models remain unchanged. To define answers to queries from $\mathcal{L}_1$ we need the following:

Definition 5.1. Let D be a domain description in $\mathcal{L}_1$ and $M = (\Psi, \Sigma)$ be an interpretation of D. We say that a *hypothesis* (2) *is true in an interpretation* M if f is true in $\Psi(\Sigma(s) \circ [a_1, \ldots, a_n])$.
Truth of arbitrary queries and sets of queries in $\mathcal{L}_1$ is defined as usual.

A reasoner with knowledge formulated in a domain description D can use hypothetical queries for various purposes. Probably, the most important of them is planning. Suppose that our reasoning agent is given a domain description D and a collection of fluent literals G viewed as a goal to be achieved by performing actions from D. Hypothetical statements of $\mathcal{L}_1$ and the entailment relation of $\mathcal{L}_1$ can be used to define a notion of a plan:

Definition 5.2. Let D be a domain description in $\mathcal{L}_1$ and G be a set of fluent literals. A sequence α of actions is a *plan* for achieving a goal G if $D \models f \textbf{ after } \alpha$ for every fluent literal $f \in G$.

A good planning program should be able to generate sequences of actions viewed as possible plans and test them using the entailment relation of $\mathcal{L}_1$. Consider for instance the following example.

Example 5.1. [Planning] Given the domain description D_4 described below

$$D_4 = \begin{cases} Facts: \\ \quad (f_1) \quad alive \ \mathbf{at} \ s_0 \\ Laws: \\ \quad (l_1) \quad shoot \ \mathbf{causes} \ \neg alive \ \mathbf{if} \ loaded \\ \quad (l_2) \quad load \ \mathbf{causes} \ loaded \end{cases}$$

and a goal $G = \{\neg alive\}$, a reasoner can come up with a candidate plan $[load, shoot]$ for achieving this goal. The candidate plan will be tested by proving a (hypothetical) statement "$\neg alive$ **after** $[load, shoot]$," i.e., by checking whether $D_4 \models \neg alive$ **after** $[load, shoot]$. It is easy to check that the statement is true and therefore $[load, shoot]$ is indeed a plan for G. Notice that if the plan is carried out in the absence of outside interference, the resulting domain description (obtained from D_4 by adding $[load, shoot]$ **occurs_at** s_0) entails G **at** s_N. Otherwise (i.e., in the presence of outside interference) the plan may be invalidated and the reasoner will be forced to continue planning from the (new) current situation. Such a re-planning will be discussed in the next section.

The previous example illustrates hypothetical reasoning about the future. In the following example we demonstrate how to perform hypothetical reasoning by using assumptions about the past.

Example 5.2. Consider the domain description D_4 and notice that it contains no information on the gun being loaded in the initial situation. Suppose now, that, given the domain description D_4, a reasoner would like to know if Fred would be dead after shooting under the assumption that initially the gun is loaded. This can naturally be represented by the query

$$q_1 = (loaded \ \mathbf{at} \ s_0) \supset (\neg alive \ \mathbf{after} \ [shoot] \ \mathbf{at} \ s_0)$$

It is easy to see that D_4 answer to q_1 is *yes*, which is, of course, the intended answer.

The entailment relation of $\mathcal{L}_1$ also allows to model more sophisticated forms of hypothetical reasoning. Notice that our translation of the informal question in the above example maps the *"if .. then"* structure of the question into a material implication $\supset$. This is possible since the premise of q_1 is possible i.e., there are models of D_4 satisfying the premise. This is not the case in the next example, where the query has a form of a counterfactual.

Example 5.3. [Counterfactuals] Consider a domain description

$$D_5 = \begin{cases} Facts: \\ \quad (f_1) \quad alive \ \mathbf{at} \ s_0 \\ \quad (f_2) \quad loaded \ \mathbf{at} \ s_0 \\ \quad (f_3) \quad unload \ \mathbf{occurs\_at} \ s_0 \\ Laws: \\ \quad (l_1) \quad shoot \ \mathbf{causes} \ \neg alive \ \mathbf{if} \ loaded \\ \quad (l_2) \quad load \ \mathbf{causes} \ loaded \\ \quad (l_3) \quad unload \ \mathbf{causes} \ \neg loaded \end{cases}$$

Given this information the reasoner must be able to conclude that "If someone used the gun to shoot the turkey at point zero then the turkey would be dead." This is an example of a counterfactual statement (no shooting actually occurred at the situation s_0) and as such cannot be translated by a material implication.
It can however be represented in our language as an atomic hypothetical statement

$$q_2 = \neg alive \ \textbf{after} \ shoot \ \textbf{at} \ s_0$$

It is easy to see that, in accordance with our expectations, $D_5 \models q_2$. Using the same type of translation one can check that the statement "If someone used the gun to shoot the turkey at point zero it would be alive" is false.

As it was mentioned above, the counterfactual character of q_2 follows from the fact that according to D_5 no shooting actually occurred at s_0 and hence q_2 is not only hypothetical but also contrary to the occurrence fact (f_3)[10].

Let us now consider a statement "If at the initial moment the gun were not loaded and someone used it to shoot the turkey then the turkey would not be dead." This is a counterfactual also, but this time its "if" part is contrary not only to the occurrence fact (f_3) as in the previous example but also to the fluent fact (f_2). It is not difficult to see that intuitively this statement is true in D_5. To represent it in our language we need to represent a state of the world which is closest to s_0 and in which the gun is unloaded. It is easy to see that this state differs from s_0 only by the value of fluent *loaded* and can be achieved by executing the action *unload*. Using this idea we can represent the above query as[11]:

$$q_3 = alive \ \textbf{after} \ [unload, shoot] \ \textbf{at} \ s_0$$

and check that D_5 indeed entails q_3.

These examples, of course, are only meant to illustrate the power of $\mathcal{L}_1$ and its entailment relation. A detailed study of hypothetical and counterfactual reasoning in $\mathcal{L}_1$ and its extensions is beyond the scope of this paper.

Finally, the following definitions and notations can be useful in our further discussions. Let H_1 and H_2 be two sets of hypotheses. We say that the premise H_1 entails conclusion H_2 in D if H_2 is true in every model of D in which H_1 is true. We will denote this by $H_1 \models_D H_2$. It is easy to see that the following result holds.

Proposition 5.1. $\emptyset \models_D H$ *iff* $D \models H$. □

A set of hypotheses H is *inconsistent* w.r.t. a domain description D if no model of D satisfies H.

It is important to notice that the entailment relation ($\models_D$) defined by D is *monotonic* —addition of new hypotheses to H_1 can only decrease the set of models of D satisfying H_1 and hence can only increase the set of conclusions. Non-monotonicity can occur only when new factual information about the world (i.e., new laws or new facts) are added to a reasoner's knowledge.

[10] Notice, that though q_2 is also expressible in $\mathcal{A}$ as well asin situation calculus, $f_3 = unload$ **occurs_at** s_0 is not expressible in either and hence q_2 loses its counterfactual character when represented in $\mathcal{A}$ or in situation calculus.

[11] In general for any fluent literal f we can introduce an action a_f and a causal rule a_f **causes** f and translate a counterfactual of the form "if f were true at s then g would have been true at s" as g **after** a_f **at** s.

6. AN ARCHITECTURE FOR AUTONOMOUS AGENTS

The ability to express the current situation, record facts and perform hypothetical reasoning makes $\mathcal{L}_1$ a viable candidate for use in designing intelligent agents capable of planning and acting in a changing environment.

In this section we outline a possible approach to such a design. In contrast with previous sections, the material covered here will have a somewhat speculative character —we briefly describe several modules comprising the agent control program and illustrate the main algorithm through examples. In this we will be heavily relying on the reader's intuition. The next sections, however, contain a mathematical description of one of these modules together with a provenly correct (but domain independent and hence inefficient) Prolog implementation.

Let us assume that our agent is capable of observing (or otherwise receiving from the outside world) the values of fluents and occurrences of actions as well as of executing some actions. We also assume that it is maintaining and using its own internal model of the world and that it has a collection of (possibly prioritized) goals which can be updated dynamically from the outside. The agent's behavior is controlled by the following simple loop:

1. The agent examines the outside world and stores the obtained information in its internal model;
2. the most urgent goals are selected from the set of goals (possibly including those obtained in step 1.;
3. the agent constructs a plan to achieve these goals and executes one or more actions of this plan. The information about this execution is stored in the agent's internal model of the world and the control goes back to the step one.

The above architecture makes many simplifying assumptions. It assumes, for instance, that the time used by the agent for finding a plan and executing an action is sufficiently short to avoid disruption, that observations made by the agent are always correct, etc. Further research will establish how restrictive these assumptions really are. But even in the presence of these assumptions the task of building (or even modeling) our intelligent agent is a difficult one. In particular, to mechanize the above control strategy we need (among other things) to have a precise, unambiguous language suitable for describing the agent's internal model. We believe that $\mathcal{L}_1$ can be successfully used for this purpose[12]. To illustrate this point, let us consider again the agent John (from Section 1) trying to achieve his goals in a changing environment.

Example 6.1. Consider the story about John from Section 1. We assume that John can execute actions *pack, drive, rent* and observe the truth values of fluents *home, at_airport, has_car*, and *packed* and occurrences of the action *hit*.

Initially, the internal world model of John contains a description of the effects of these actions which can be represented by the following domain description in $\mathcal{L}_1$:

[12] One important feature needed for this purpose which is not available in $\mathcal{L}_1$ is the ability to represent and reason about concurrent actions. This restriction is only imposed however to simplify the presentation and can be easily lifted (see [BG93]).

$$D_8 = \begin{cases} \textit{Laws}: \\ (l_1)\ \textit{rent}\ \textbf{causes}\ \textit{has\_car} \\ (l_2)\ \textit{hit}\ \textbf{causes}\ \neg \textit{has\_car} \\ (l_3)\ \textit{drive}\ \textbf{causes}\ \textit{at\_airport}\ \textbf{if}\ \textit{has\_car} \\ (l_4)\ \textit{drive}\ \textbf{causes}\ \neg \textit{home}\ \textbf{if}\ \textit{has\_car} \\ (l_5)\ \textit{pack}\ \textbf{causes}\ \textit{packed}\ \textbf{if}\ \textit{home} \end{cases}$$

John's initial observation of values of fluents can be recorded by the following collection of facts:

$$F_0 = \begin{cases} \textit{facts}: \\ (f_1)\ \textit{home}\ \textbf{at}\ s_0 \\ (f_2)\ \neg \textit{at\_airport}\ \textbf{at}\ s_0 \\ (f_3)\ \textit{has\_car}\ \textbf{at}\ s_0 \end{cases}$$

His goal of bringing a packed suitcase to the airport has the form

$\textbf{currently}\ (\textit{packed} \wedge \textit{at\_airport})$.

To find a plan of actions John can use Definition 2 from the previous section, i.e., he needs to search for a sequence α of actions such that[13]

$J_0 \models (\textit{packed} \wedge \textit{at\_airport})\ \textbf{after}\ \alpha$

where $J_0 = D_8 \cup F_0$. It is easy to check that $s_N = s_0$ and

$J_0 \models (\textit{packed} \wedge \textit{at\_airport})\ \textbf{after}\ \alpha_0$

where $\alpha_0 = [\textit{pack}, \textit{drive}]$. Satisfied with the plan, John packs his suitcase. The execution of this action is recorded by expanding J_0 by the new facts

(f_4) $\quad \textit{pack}\ \textbf{occurs\_at}\ s_0$

(f_5) $\quad s_0\ \textbf{precedes}\ s_1$

Let us denote the resulting description by J_1. All he needs to do now is to execute $\alpha_1 = [\textit{drive}]$. Suppose however, that only one action can be executed between the observations, and therefore John's control goes back to Step 1. in which he observes his car being hit by a truck. This event is recorded by expanding J_1 by the statements

(f_6) $\quad \textit{hit}\ \textbf{occurs\_at}\ s_1$

(f_7) $\quad s_1\ \textbf{precedes}\ s_2$

It is easy to see that for the resulting domain description J_2

$J_2 \models\ \textbf{currently}\ \neg \textit{has\_car}$

and

[13]Here and in the rest of the paper we use $f_1 \wedge \ldots \wedge f_n$ **after** α **at** S as an abbreviation for $(f_1$ **after** α **at** $S) \wedge \ldots \wedge (f_n$ **after** α **at** $S)$.

$J_2 \not\models (packed \wedge at\_airport)$ **after** α_1

and hence the plan α_1 is invalidated by this new information. To revise the plan, John poses the query

$?\!\!-\!\!(packed \wedge at\_airport)$ **after** α

to J_2. It is again easy to check that

$J_2 \models (packed \wedge at\_airport)$ **after** α_2

where $\alpha_2 = [rent, drive]$.

Suppose now that John goes on to execute α_2 (this time without unpleasant interruptions). It is easy to see that the resulting domain description obtained from J_2 by adding the statements

(f_8)	*rent* **occurs_at** s_2
(f_9)	s_2 **precedes** s_3
(f_{10})	*drive* **occurs_at** s_3
(f_{11})	s_3 **precedes** s_4

entails John's goal.

The above discussion suggests that the agent's program can be constructed from four modules, *observe*, *select_goal*, *plan*, and *execute* organized in a simple loop. To further clarify our intuition we give a short description of these modules.

- The module *observe* consists of two parts: *observe_facts* and *obtain_goals*. The former is responsible for obtaining new fluent and occurrence facts from the outside world and storing them in the agent's internal model D. The latter gets new goals together with their priorities and adds them to the set of goals of the agent. The module *observe_facts* considers the current internal model D with current situation s_k and performs the following three steps:
  - It calls a submodule $observe\_fluents(s_k)$ which checks the values of observable fluents in s_k and expands D by the corresponding fluent facts. (Notice that s_k remains the current situation of the new internal model).
  - It then calls a submodule $observe\_action(s_k)$ which checks if some action occurred at s_k.
  - If action a is observed to have occured at s_k then it again expands D by the statements

    a **occurs_at** s_k and

    s_k **precedes** s_{k+1},

    where s_{k+1} is a situation constant not occurring in D. It then again calls *observe_fluents* with the parameter s_{k+1}. Notice that now s_{k+1} becomes the current situation of D.
- The next module, *select_goal*, is probably the simplest. It selects the most urgent goals from the set of all goals of the agent.

- The module *plan* takes the set of fluent literals from the goal g selected in the previous step and searches for a sequence α of actions such that for all f in this set, $D \models f$ **after** α. If the agent is capable of efficiently testing the entailment relation of $\mathcal{L}_1$, such an α can be found by simple generate-and-test methods. We will call such planners $\mathcal{L}$*—planners*. Despite their simplicity, they have several attractive features which make them an interesting subject for investigation. They can be used to compare expressive power and efficiency of various nonmonotonic formalisms used to do the testing or serve as a testbed for developing mathematical methods for proving properties of planners, such as correctness and completeness and stability of the produced plans in a dynamic world among others. They can also be instrumental in evaluating efficiency of various domain dependent heuristics controlling the "generate" part of the program, comparing different approaches to planning, etc. Some of the points mentioned above will be elaborated upon in the next two sections.
- Finally, the module *execute* consists of the (possibly) complex part responsible for physical execution of an action a and a simple "bookkeeping" part which expands the internal model D of the agent by the two statements

  a **occurs_at** s_k

  s_k **precedes** s_{k+1}

  where s_k is the current situation of D and s_{k+1} is a situation constant not occurring in D. For this description to be valid, D must contain the unique current situation, i.e., there must be a situation constant s_k in D s.t. $D \models (s_k = s_N)$. It is not difficult to see that our algorithm guarantees this.

We hope that the informal discussion in this section could convince the reader that the ability to express the current situation, record facts and do hypothetical reasoning makes $\mathcal{L}_1$ an interesting candidate for a language of a reasoning agent's internal model. If nothing else it certainly can serve as a powerful specification language allowing rather concise description of the desired behavior of an agent.

Much more work, of course, is needed to check if such an agent architecture can be efficiently implemented. Prospects of actual implementation critically depend on our understanding of the entailment relation of $\mathcal{L}_1$ and the ability to automate this relation. The next two sections contain a first step in this direction.

7. APPROXIMATING THE ENTAILMENT RELATION OF $\mathcal{L}_1$

Our methodology for computing the entailment relation of $\mathcal{L}_1$ is based on translating a domain description D of $\mathcal{L}_1$ into a declarative logical theory Π_D approximating the entailment of $\mathcal{L}_1$ in D from below and using a general purpose inference mechanism for Π_D to answer queries in D. The success of this approach depends critically on our ability to construct a Π_D which is a good (complete or almost complete) approximation of D and on the efficiency of the inference mechanism of Π_D. Most of the work related to computing the entailment relations of action description languages concentrated on the former and hence required rather powerful logics such as circumscription, disjunctive or abductive logic programming, etc. In contrast, in this paper we will concentrate on the latter (for sound and complete translation of $\mathcal{L}_1$ into circumscription see [BGP96]).

Our translation Π_D of D will be a declarative logic program (under answer set semantics) used together with the inference mechanism incorporated in the standard Prolog interpreter (as implemented, for instance, in Quintus Prolog). It is well known that in general this mechanism is not complete and —due to the problems with floundering and absence of the occur check— may even be unsound. We show however that in our particular case this is not a problem, since for any translation Π_D of a domain description D obtained by the algorithm from the previous section the Prolog interpreter is shown to be sound and complete w.r.t. Π_D's declarative semantics (see Corollary 7.1 below).

We pay, of course, for the choice of the weak language, by a possible incompleteness of Π_D w.r.t. D. But even from this standpoint the situation is not as bad as one might expect. In general, there are two sources of incompleteness of Π_D: lack of information about the values of fluents in the initial situation and lack of information about occurrences of actions and temporal relations between situations. Fortunately, domain descriptions built by agents constructed according to our architecture have "explicit actual paths" and hence allow only incompleteness of knowledge about values of fluents. The following definition clarifies the term.

Definition 7.1. Let D be a consistent domain description and $s_0, \ldots, s_k$ be a list of situation constants occurring in statements from D. We will say that D has an *explicit actual path* if

(a) $(s_i \textbf{ precedes } s_{i+1}) \in D$ $(0 \leq i < k)$;
(b) $D \models s_k = s_N$;
(c) there is a sequence $\alpha = [a_0, \ldots, a_{k-1}]$ of actions such that:

- $(a_i \textbf{ occurs\_at } s_i) \in D$ $(0 \leq i < k)$ and
- $D \models (\alpha \textbf{ occurs\_at } s_0)$.

It is easy to see that any D satisfying these conditions has a unique actual path; i.e., for any model (Ψ, Σ) of D and for every $0 < i \leq k$, $\Sigma(s_i) = [a_0, \ldots, a_{i-1}]$ and $\Sigma(s_N) = \Sigma(s_k) = \alpha$.

Definition 7.2. We will say that a domain description D is *simple* if

1. D is consistent;
2. D has an explicit actual path;
3. all facts of D are atomic and
4. D does not contain contradictory causal laws.

The last assumption in the definition is not essential. Its purpose is to make Π_D simpler. The assumption can be removed by adding rules to Π_D which define the executability of an action in a situation.

Now, let us describe a program Π_D approximating the entailment relation of D. The degree of incompleteness of Π_D will depend only on the amount of information about the values of fluents in the initial situation lost in the process of translation.

We will show that if our knowledge of these values is complete then so is the translation. In our further work we plan to investigate the use of abduction for narrowing the gap between D and Π_D.

7.1. Logic programming approximation of D

Let us proceed with the construction of logic programming approximation Π_D of D. The alphabet of Π_D will consist of symbols for actions, fluent literals and situations from the language of D, predicates *true_at, true_after, all_true_after, false_after, one_false_after, occurs_at, causes, imm_follows, current, contrary, member, fluent_literal* and *ab,* and a standard Prolog function symbol $[FirstEl \mid RestOfList]$ for creating a list. We will use the following typed variables:

A	for actions
R	for lists of actions
F, G	for fluent literals
P	for lists of fluent literals

All the variables can be indexed. The corresponding lower case letters will denote constants of respective types. Predicate symbol *false_after* can be viewed as negation of *true_after*. To reflect this agreement we introduce an auxiliary terminology. We say that atoms $true\_after(\bar{t})$ and $false\_after(\bar{t})$ (where $\bar{t}$ is a list of terms) are *incompatible* and that a program Π in the above language is *consistent* if it has an answer set and if no answer set of Π contains incompatible atoms.

Let D be a simple domain description with the explicit actual path $a_0, \ldots, a_{k-1}$. The logic programming approximation Π_D of entailment of D will consist of the following rules:

1. **Domain Dependent Axioms**

   (a) **(AP) Description of Actual Path**

   $imm\_follows(s_1, s_0).$
   $\ldots$
   $imm\_follows(s_k, s_{k-1}).$

   $occurs\_at(a_0, s_0).$
   $\ldots$
   $occurs\_at(a_{k-1}, s_{k-1}).$

   (b) **(BC) Boundary Conditions**

   $true\_at(f, s_i) \in \Pi_D$ for each $(f \textbf{ at } s_i) \in D$

   (c) **(CL) Causal Laws**

   $causes(a, f, p) \in \Pi_D$ for each $(a \textbf{ causes } f \textbf{ if } p) \in D$

2. **Domain Independent Axioms**

   (a) **(EA) Effects of actions**

$$
\begin{array}{llll}
e_1. & true\_after(F,[\,],S) & :- & true\_at(F,S). \\
e_2. & true\_after(F,[A|R],S) & :- & causes(A,F,P), \\
 & & & all\_true\_after(P,R,S). \\
e_3. & false\_after(F,R,S) & :- & contrary(F,G), \\
 & & & true\_after(G,R,S). \\
e_4. & all\_true\_after([\,],R,S). & & \\
e_5. & all\_true\_after([F|P],R,S) & :- & true\_after(F,R,S), \\
 & & & all\_true\_after(P,R,S). \\
e_6. & one\_false\_after(P,R,S) & :- & member(F,P), \\
 & & & false\_after(F,R,S).
\end{array}
$$

The first three axioms describe effects of actions on individual fluents while axioms (e_4)—(e_6) define truth and falsity for lists of fluent literals.

(b) **(FI) First Inertia Axiom**

$$
\begin{array}{llll}
i_1. & true\_after(F,[A|R],S) & :- & fluent\_literal(F), \\
 & & & true\_after(F,R,S), \\
 & & & not\ ab(F,A,R,S).
\end{array}
$$

$$
\begin{array}{llll}
i_2. & ab(F,A,R,S) & :- & contrary(F,G), \\
 & & & causes(A,G,P), \\
 & & & not\ one\_false\_after(P,R,S).
\end{array}
$$

(c) **(SI) Second Inertia Axiom**

$$
\begin{array}{lll}
true\_at(F,S2) & :- & fluent\_literal(F), \\
 & & imm\_follows(S2,S1), \\
 & & occurs\_at(A1,S1), \\
 & & true\_after(F,[A1],S1).
\end{array}
$$

3. **Library Axioms**

(a) **Contrary**[14]

$contrary(\neg F, F).$
$contrary(F, \neg F).$

(b) **Member**

$member(X,[X|\_]).$
$member(X,[\_|T])\ :- member(X,T).$

Notice that relation *fluent_literal(F)* is not defined in Π_D. We will assume that it is given by a list of atoms not containing "[ ]" and, hence, the truth of lists of fluents can only be established by axioms (e_4) and (e_5).

It is important to realize that at this stage the rules of Π_D are ground instantiations of the above schemas where substitutions of variables by ground terms are done in accordance with definitions of their types. This means that occurrences of *fluent_literal* in the premises of the Inertia axioms are redundant and can be removed without any effect on the declarative meaning of the program. They will be needed though to ensure correctness of answers to queries given by the Prolog interpreter.

[14] The notation $\neg F$ denotes the term $neg(F)$.

7.2. Soundness and Completeness of Π_D entailment

In this subsection we formulate and show the soundness of Π_D with respect to D for simple domain descriptions. We also show that for a restricted class Π_D is complete with respect to D.

Since $D \models f$ **at** s iff $D \models f$ **after** [] **at** s, from now on we will limit our query language to formulas of the form f **after** α **at** s. For any query q of this form, by $\pi(q)$ we will denote $true\_after(f, \alpha, s)$.

We will say that a (declarative) program Π_D is *sound* w.r.t. D if Π_D is consistent and for any query q, if $\Pi_D \models \pi(q)$ then $D \models q$. Let us now prove the soundness of Π_D. This will guarantee that the entailment relation of Π_D can be used to approximate the entailment of D. Before we state the theorem, we will show using the following example that Π_D is not always complete.

Example 7.1. Consider the domain description:

$$D_{11} = \begin{cases} Facts: \\ \ (f_1) \quad a \ \textbf{occurs\_at} \ s_0 \\ \ (f_2) \quad s_0 \ \textbf{precedes} \ s_1 \\ Laws: \\ \ (l_1) \quad a \ \textbf{causes} \ f \ \textbf{if} \ g \\ \ (l_2) \quad a \ \textbf{causes} \ f \ \textbf{if} \ \neg g \end{cases}$$

Clearly, $D_{11} \models f$ **at** s_1, while $\Pi_{D_{11}}$ does not entail[15] $true\_after(f, [\,], s_1)$.

The proof of the soundness theorem is based on the following lemmata. First, we will introduce some notation. Consider ground instantiations of all the rules of Π_D except the Second Inertia Axiom (SI) and the Description of Actual Path (AP). The set of all such instantiations not containing any other situation constants except s_i will be denoted by H_i. It is easy to see that Π_D consists of the union of the sets $H_0, \ldots, H_k$ together with ground instantiations of SI and AP.

Lemma 7.1. Consider a simple domain description D. Let Π_1 be obtained from Π_D by replacing (SI) and (AP) by

$true\_at(F, s_i)$:– $true\_after(F, [a_{i-1}], s_{i-1})$.

where $0 < i \leq k$ and (a_{i-1} **occurs_at** *$s_{i-1}) \in D$).*

Then, for any query q in D, $\Pi_1 \models \pi(q)$ iff $\Pi_D \models \pi(q)$.

PROOF. Follows immediately from the Splitting Lemma (see Lemma □

Lemma 7.2. Let D be a simple domain description. For any $0 \leq i \leq k$ and any collection I of formulas of the form $true\_at(f, s_i)$ s.t. $D \models f$ **at** *s_i the program $H_i \cup I$ has unique consistent answer set and is sound w.r.t. D.*

PROOF. It is easy to see that $H_i \cup I$ is acyclicand hence it has a unique answer set [AB90]. Let us denote it by $\mathcal{S}_i$. We prove the soundness and consistency of $H_i \cup I$ by showing that for any f, r, s_i, p:

[15] To obtain completeness we can expand our program by the axiom "$true\_at(g, s_0)$ or $true\_at(\neg g, s_0)$" (where *or* is the epistemic disjunction [GL91]) or use abduction. This approach works in general too but its usefulness depends somewhat on the development and availability of query answering systems for disjunctive and abductive programs.

$(i)\quad true\_after(f, r, s_i) \in \mathcal{S}_i \;\Rightarrow\; D \models f \textbf{ after } r \textbf{ at } s_i$

$(ii)\quad all\_true\_after(p, r, s_i) \in \mathcal{S}_i \;\Rightarrow\; \forall f \in p.\ D \models f \textbf{ after } r \textbf{ at } s_i$

$(iii)\quad true\_after(f, r, s_i) \notin \mathcal{S}_i \textit{ or } false\_after(f, r, s_i) \notin \mathcal{S}_i$

$(iv)\quad one\_false\_after(p, r, s_i) \notin \mathcal{S}_i \textit{ or } \exists f \in p.\ true\_after(f, r, s_i) \notin \mathcal{S}_i$

The statements (i) and (ii) guarantee soundness and the statements (iii) and (iv) guarantee consistency of $H_i \cup I$.

Let us use induction on the length of r in formulae (i)—(iv).

Base Case: length of r is 0.

(i.0) $true\_after(f, [\,], s_i) \in \mathcal{S}_i$
iff
$true\_at(f, s_i) \in \mathcal{S}_i$ (e1)
iff
$true\_at(f, s_i) \in I$ or $true\_at(f, s_i) \in H_i$
implies
$D \models f \textbf{ at } s_i$.

(ii.0) (By induction on the length of p)
base: length of p is 0 (vacuously true)

inductive hypothesis: If length of p is less than m then $all\_true\_after(p, [\,], s_i) \in \mathcal{S}_i$
implies
for all f in p, $D \models f \textbf{ after } [\,] \textbf{ at } s_i$.

induction: Since length of p is greater than zero there exists a fluent g and a (possibly empty) list of fluents p' such that $p = [g|p']$. Then

$all\_true\_after(p, [\,], s_i) \in \mathcal{S}_i$
iff
$true\_after(g, [\,], s_i) \in \mathcal{S}_i$ and $all\_true\_after(p', [], s_i) \in \mathcal{S}_i$ (e_5)
implies
$D \models g \textbf{ after } [\,] \textbf{ at } s_i$ and
for all f in p', $D \models f \textbf{ after } [\,] \textbf{ at } s_i$ ((i.0) and inductive hypothesis).
implies
for all f in p, $D \models f \textbf{ after } [\,] \textbf{ at } s_i$. (since $p = [g|p']$)

(iii.0) Since D is a consistent domain description:

$$D \not\models f \textbf{ at } s_i \textit{ or } D \not\models \neg f \textbf{ at } s_i$$

i.e.,

$$true\_at(f, s_i) \notin H_i \cup I \textit{ or } true\_at(\neg f, s_i) \notin H_i \cup I$$

Since there are no other rules in $H_i \cup I$ with heads formed by the predicate $true\_at$, we have that $true\_at(f, s_i) \notin \mathcal{S}_i$ or $true\_at(\neg f, s_i) \notin \mathcal{S}_i$. Hence, by e_1 and e_3, we obtain:

$$true\_after(f,[\,],s_i) \notin \mathcal{S}_i \ or \ false\_after(f,[\,],s_i) \notin \mathcal{S}_i.$$

(iv.0) (by induction on the length of p)
base: length of p is 0.
The only rule with one_false_after in the head whose first argument is a list is e6. But when the first argument of one_false_after is [ ], no ground instance of the body of e6 belongs to $\mathcal{S}_i$ (since there is no g such that $member(g,[\,]) \in \mathcal{S}_i$) and hence $one\_false\_after([\,],[\,],s_i) \notin \mathcal{S}_i$.

inductive hypothesis:
If the length of p is less than m, then either $one\_false\_after(p,[\,],s_i) \notin \mathcal{S}_i$ or there exists an f in p such that $true\_after(f,[\,],s_i) \notin \mathcal{S}_i$.

induction: Since length of p is greater than zero there exists a fluent g and a (possibly empty) list of fluents p' such that $p = [g|p']$.
By inductive hypothesis, either
(a_0) $one\_false\_after(p',[\,],s_i) \notin \mathcal{S}_i$ or
(b_0) there exists an f in p' s. t. $true\_after(f,[\,],s_i) \notin \mathcal{S}_i$.
If case (b_0) is true then we are finished, since $p' \subset p$.
Now, consider case (a_0); by using (iii.0) we have that either

(a_1) $false\_after(g,[\,],s_i) \notin \mathcal{S}_i$ or
(b_1) $true\_after(g,[\,],s_i) \notin \mathcal{S}_i$.

If case (b_1) is true then we are done since $g \in p$.
Now consider the case when (a_0) and (a_1) are true.
Using e_6, Lemma .1 (from Appendix 9) and (a_0) we have that there is no $f \in p'$ such that $false\_after(f,[\,],s_i) \in \mathcal{S}_i$. The fact that $p = [g|p']$ together with (a_1) implies that there is no $f \in p$ such that $false\_after(f,[\,],s_i) \in \mathcal{S}_i$.
As a result, for no instance of (e_6) with $one\_false\_after(p,[\,],s_i)$ in the head its body belongs to $\mathcal{S}_i$ and hence $one\_false\_after(p,[\,],s_i) \notin \mathcal{S}_i$ follows immediately from by Lemma .1 in.

Inductive Step: Suppose (i)—(iv) are true when the length of r is less than n. Let us prove that it is true when length of r is n.

Since length of r is greater than zero, let us assume $r = [a|r']$.

(i.n) $true\_after(f,r,s_i) \in \mathcal{S}_i$
iff
(case a) $(causes(a,f,p) \in \mathcal{S}_i$ and $all\_true\_after(p,r',s_i) \in \mathcal{S}_i)$ or
(case b) $(true\_after(f,r',s_i) \in \mathcal{S}_i)$ and $ab(f,a,r',s_i) \notin \mathcal{S}_i$. (Using e_2, i_1 and Lemma .1)
(case a) From the inductive hypothesis we have that $all\_true\_after(p,r',s_i) \in \mathcal{S}_i)$ implies
for all f in p, $D \models f$ **after** r' **at** s_i.
It is easy to see that $causes(a,f,p) \in \mathcal{S}_i$ iff (a **causes** f **if** p) $\in D$.
Since our domain descriptions do not contain contradictory causal laws, a is executable after the execution of r' in the situation s_i. Hence,f **after** r **at** s_i must be true in all models of D. Therefore, $D \models f$ **after** r **at** s_i.
(case b)From the inductive hypothesis we have $true\_after(f,r',s_i) \in \mathcal{S}_i)$ implies

$D \models f$ **after** r' **at** s_i. $(i.n.b.1)$
$ab(f, a, r', s_i) \notin \mathcal{S}_i$
iff
for any pair a and p, if $causes(a, \overline{f}, p) \in \mathcal{S}_i$ then $one\_false\_after(p, r', s_i) \in \mathcal{S}_i$ (from i_2 and Lemma .1)
iff
If (a **causes** $\overline{f}$ **if** p) $\in D$ then for some g in p, $true\_after(\overline{g}, r', s_i) \in \mathcal{S}_i$ (Using e_6, e_3 and Lemma .1)
iff
If (a **causes** $\overline{f}$ **if** p) $\in D$ then for some g in p, $D \models \overline{g}$ **after** r' **at** s_i (Using inductive hypothesis.) $(i.n.b.2)$
If f is a positive fluent literal then $(i.n.b.1)$ implies that $f \in \Psi(\Sigma(s_i) \circ r')$, and $(i.n.b.2)$ implies that $f \notin E^-_a(\Psi(\Sigma(s_i) \circ r'))$. i.e., $f \in \Psi(\Sigma(s_i) \circ r' \circ a)$. That means $D \models f$ **after** r **at** s_i. The reasoning when f is a negative fluent literal is similar.

(ii.n) The proof is the same (modulo replacing [ ] by r) as the proof in ii.0.

(iii.n) We will show that $true\_after(f, r, s_i) \in \mathcal{S}_i$ and $false\_after(f, r, s_i) \in \mathcal{S}_i$ can not be true at the same time.
Suppose $true\_after(f, r, s_i) \in \mathcal{S}_i$. ¿From (i.n) we have $D \models f$ **after** r **at** s_i. (iii.n.1) Now, suppose $false\_after(f, r, s_i) \in \mathcal{S}_i$. Using e_3 and Lemma .1 we have $true\_after(\overline{f}, r, s_i) \in \mathcal{S}_i$. Using (i_n) we will have $D \models \overline{f}$ **after** r **at** s_i. $(iii.n.2)$
Since D is consistent we can not have both $(iii.n.1)$ and $(iii.n.2)$ to be true. Hence, we have a contradiction. i.e., our initial assumption was false. Finally, either $true\_after(f, r, s_i) \notin \mathcal{S}_i$ or $false\_after(f, r, s_i) \notin \mathcal{S}_i$.

(iv.n) The proof is the same (modulo replacing [ ] by r) as the proof in $iv.0$. □

Now we can move forward to *re-introduce* a connection between situations actually following one another and prove an equiovalent of the Lemma above. To this pourpose, let l_i $(0 < i \leq k)$ be the rule

$true\_at(f, s_i) :\!-true\_after(f, [a_{i-1}], s_{i-1})$

Lemma 7.3. For any $0 \leq m \leq k$, the program T_m:

$T_m = H_0 \cup (l_1 \cup H_1) \cup \ldots \cup (l_m \cup H_m)$

has a unique answer set and is sound w.r.t. D.
Proof. By induction on m.

<u>Base Case</u>: $m = 0$. Conclusion of the lemma follows immediately from Lemma 7.2.

<u>Inductive step</u>: Obviously, $T_m = T_{m-1} \cup (l_m \cup H_m)$. By inductive hypotheses T_{m-1} has a unique consistent answer set B_{m-1}. Notice, that the set $U = head(T_{m-1})$ (see Appendix 9) forms a splitting [LT94] set for T_m and hence

1. B_m is an answer set of T_m iff $B_m = B_{m-1} \cup C$ where C is an answer set of the partial evaluation $e_U(l_m \cup H_m, B_{m-1})$ of $(l_m \cup H_m)$ w.r.t. B_{m-1}. Also,
2. $e_U(l_m \cup H_m, B_{m-1}) = H_m \cup I_m$ where

$I_m = \{true\_at(f, s_m) \ : \ T_{m-1} \models true\_after(f, [a_{m-1}], s_{m-1})\}$.

By inductive hypotheses T_{m-1} is sound w.r.t. D and hence 2. implies that for any fluent literal f s.t. $true\_at(f, s_m) \in I_m$, $D \models true\_after(f, [a_{m-1}], s_{m-1})$. Let $\Sigma(s_k)$ be the actual path of D. Then $\Sigma(s_m) = \Sigma(s_{m-1}) \circ a_{m-1}$ and hence $D \models f$ **at** s_m.
Therefore, I_m satisfies the conditions of Lemma 7.2 and hence $I_m \cup H_m$ have unique consistent answer set C and is sound w.r.t. D. This, together with 1. above, implies the conclusion of Lemma 7.3 for T_m. □

Theorem 7.1 (Soundness of Π_D). For any simple domain description D its logic programming approximation Π_D is sound w.r.t. D.
PROOF. It is easy to see that T_k as defined in Lemma 7.3 is the same as Π_1. Hence from Lemma 7.3 the program Π_1 is sound w.r.t. D. From Lemma 7.1 the program Π_1 is equivalent to Π_D. Hence, Π_D is sound w.r.t. D. □

Theorem 7.2 (Restricted Soundness and Completeness). For any simple domain description D with explicit complete information about the initial state (i.e., for any fluent f in the language either f **at** *s_0 or $\neg f$* **at** *s_0 is in D) its logic programming approximation Π_D is sound and complete w.r.t. D.*
PROOF. It is easy to see that D has a unique model M and from Lemmata 7.1, 7.2 and 7.3, Π_D has a unique answer set $\mathcal{S}$. Since Theorem 7.1 proves the soundness we only need to prove the completeness. To prove the completeness it suffices to prove Lemma 7.4 below. □

Lemma 7.4. Let D be a simple domain description with explicit complete information about the initial state and Π_D be its logic programming approximation. Let Π_1 be as defined in Lemma 7.1. Then $D \models f$ **after** *r* **at** *s_i implies $\Pi_1 \models true\_after(f, r, s_i)$.*
PROOF. Notice that D has a unique model $M = (\Psi, \Sigma)$ and Π_1 has a unique answer set $\mathcal{S}$. We now use induction on the indexes of situations[16].

<u>Base Case</u>: index is 0.

Suppose $D \models f$ **after** r **at** s_o. We need to show that $true\_after(f, r, s_0) \in \mathcal{S}$. We prove this by induction on the length of r.

base case: length of r is 0.

$D \models f$ **after** [ ] **at** s_0
iff
$D \models f$ **at** s_0
iff
f **at** $s_0 \in D$ (Since D is consistent and has complete information about the initial state.)
iff
$true\_at(f, s_0) \in \Pi_1$
implies
$true\_after(f, [\,], s_0) \in \mathcal{S}$

inductive hypothesis: If length of r is less that n then $D \models f$ **after** r **at** s_o implies $true\_after(f, r, s_0) \in \mathcal{S}$.

[16] For a situation s_i, by its index we mean the number i.

induction: Since length of r is greater than 0, let $r = [a|r']$.

Let $D \models f$ **after** r **at** s_o
iff
$f \in \Psi(r)$
iff
$f \in \Psi(r') \cup E^+(a, \Psi(r')) \setminus E^-(a, \Psi(r'))$
iff
(a) $f \in \Psi(r') \setminus E^-(a, \Psi(r'))$, or
(b) $f \in E^+(a, \Psi(r'))$.

(case a) It is easy to see that $f \in \Psi(r')$ iff $D \models f$ **after** r' **at** s_o
This implies (using the inductive hypothesis) that $true\_after(f, r', s_0) \in \mathcal{S}$. a_1
$f \notin E^-(a, \Psi(r'))$
iff
If (a **causes** $\overline{f}$ **if** p) $\in D$ then for some g in p, $D \models \overline{g}$ **after** r' **at** s_0
iff
If (a **causes** $\overline{f}$ **if** p) $\in D$ then for some g in p, $true\_after(\overline{g}, r', s_i) \in \mathcal{S}$ (Using inductive hypothesis)
iff
for any pair a and p, if $causes(a, \overline{f}, p) \in \mathcal{S}$ then $one\_false\_after(p, r', s_0) \in \mathcal{S}$ (Using e_6, e_3 and Lemma .1)
iff
$ab(f, a, r', s_0) \notin \mathcal{S}$ (from i_2 and Lemma .1) a.2
¿From a_1, a_2 and i1 using Lemma .1 we have that $true\_after(f, r, s_0) \in \mathcal{S}$.

(case b) $f \in E^+(a, \Psi(r'))$
iff
There is a proposition (a **causes** f **if** p) in D such that for all g in p, $D \models g$ **after** r' **at** s_0
implies
$causes(a, f, p) \in \mathcal{S}$ and for all g in p, $true\_after(g, r', s_0) \in \mathcal{S}$ (By 1.(c) in the program Π_D and using inductive hypothesis)
implies
$causes(a, f, p) \in \mathcal{S}$ and $all\_true\_after(p, r', s_0) \in \mathcal{S}$ (Using ii.0 in the Proof of Lemma 7.2 in the reverse direction.).
implies
$true\_after(f, r, s_0) \in \mathcal{S}$ (Using r_2 in the program Π_D and Lemma .1).

Inductive Step: Suppose $D \models f$ **after** r **at** s_n implies $true\_after(f, r, s_0) \in \mathcal{S}$ for all s_n, when $n < i$. Let us prove that it is true for s_i.

We prove this by induction on the length of r.

base case: length of r is 0.

$D \models f$ **after** [ ] **at** s_i
iff
$D \models f$ **at** s_i
iff
$D \models f$ **after** $[a_{i-1}]$ **at** s_{i-1}
implies
$true\_after(f, [a_{i-1}], s_{i-1}) \in \mathcal{S}$ (by inductive hypothesis)

implies
$true\_at(f, s_i) \in \mathcal{S}$ (Using l_i and Lemma .1.)
implies
$true\_after(f, [\,], s_i) \in \mathcal{S}$ (Using e_1 and Lemma .1.)

inductive hypothesis: If length of r is less that n then $D \models f$ **after** r **at** s_i implies $true\_after(f, r, s_i) \in \mathcal{S}$.

induction: We need to show that when length of r is n, $D \models f$ **after** r **at** s_i implies $true\_after(f, r, s_i) \in \mathcal{S}$.
The proof of this case is similar to that for $i = 0$. □

7.3. Computing with Π_D

We now proceed in the direction of implementing the declarative logic programs obtained by the translations. Theorem 7.1 guarantees soundness of Π_D viewed as a declarative logic program. If we want to use a computer to actually answer queries to Π_D we need to use a particular query answering algorithm. The Prolog interpreter, based on the unification algorithm and SLDNF proof procedure with leftmost selection rule [Cla78, AD94], is certainly the most popular among the large family of such algorithms and seems to be a natural choice for use in this paper. The following theorems ensure that it can be used safely.

Before formulating the result we will change our notational convention. In what follows we will stop identifying Π_D with the collection of ground instances of its rules where variables are replaced by ground terms in accordance with definitions of their sorts. Instead, Π_D should be viewed as a logic program *with variables* (used as an input by the Prolog interpreter). The collection of all its ground instances in (unsorted) language of Π_D will be denoted by Π^1_D; the collection of ground instances obtained from Π_D by substitutions honoring sorts will be denoted by Π^2_D. As before, our queries will be ground atoms in the sorted language. In the following proofs Π_D will be used mainly when we talk about Prolog inference while Π^1_D will be used when referring to the declarative semantics.

Theorem 7.3. For any simple domain description D, its logic programming approximation Π_D, and a query q the Prolog interpreter is sound w.r.t. Π^1_D, i.e., if the answer of Prolog to $\pi(q)$ is:
then $\Pi^1_D \models \pi(q)$;
yes no *then* $\Pi^1_D \not\models \pi(q)$.

PROOF. See discussion and Lemmata below. □

As mentioned above the Prolog interpreter can be viewed as implementation of SLDNF resolution with the Prolog selection rule which, at every step, selects for resolution the leftmost literal. We will assume familiarity with the notion of an SLDNF tree for a query $\pi(q)$ (w.r.t. program Π) as presented, for instance, in [AB94]. Each such tree corresponds to a possible selection rule.

If the rule always selects the leftmost literal the corresponding tree is called LDNF-tree[17]. Branches of the LDNF tree for $\pi(q)$ represent all possible LDNF derivations in $\Pi \cup \{\pi(q)\}$; their nodes contain goals – finite sequences of atoms

[17] The tree here is viewed as a directed graph and hence the ordering of branches is ignored.

possibly preceded by *not* . Each finite branch ends with a node marked by *success* (the node contains the empty query), *flounder* (the leftmost member of the node's goal is an atom preceded by *not* and containing an uninstantiated variable), or *failed* (no other resolution is possible). In general, some of the branches of the tree can, of course, be infinite.

The LDNF resolution proof procedure can be viewed as search of this tree T. For ground queries this search returns *yes* if the particular subtree T_0 (called the main tree) of T contains a branch with final node marked by success and *no* if all branches of T_0 have final nodes marked by *failed*. It is well known that the algorithm is sound w.r.t. answer set semantics, i.e., if its answer to $\pi(q)$ is *yes* then $\Pi \models \pi(q)$, if the answer is *no* then no answer set of Π contains $\pi(q)$.

The actual Prolog interpreter used in various Prolog systems can be viewed as an implementation of the LDNF proof procedure with two important exceptions:

1. it allows selection of non-ground queries of the form *not* p, i.e., floundering is ignored;
2. implementation of the unification algorithm omits the occur check and may, therefore, produce unsound results.

These observations suggest the following structure of the proof.

a) First, we demonstrate that for every ground query $\pi(q)$, $\Pi_D \cup \{\pi(q)\}$ is occur-check free [AP94], i.e., the non-deterministic unification algorithm of Martelli and Montanari [MM82] (whose variants are normally implemented in Prolog interpreters) never selects a step requiring the occur check.

b) Next, let us show that LDNF-trees for ground queries in Π_D do not have nodes marked by *flounder*.

The observations above imply that for ground queries to Π_D the differences between Prolog interpreter and LDNF proof procedure can be ignored and hence, by soundness theorem for LDNF we have that for any query $\pi(q)$ if the Prolog answer to $\pi(q)$ is *yes* then $\Pi^1_D \models \pi(q)$ and if the answer is *no* then $\Pi^1_D \not\models \pi(q)$. We proceed by proving the following lemmas.

Lemma 7.5. For a simple domain description D, the program Π_D is occur-check free w.r.t. ground queries.

PROOF. To prove the lemma we need the notion of well-moded program. The concept, due to Dembinski and Maluszynski [DM85], proved to be useful for establishing various properties of logic programs. We will need the following terminology:

By a *mode* for an n-ary predicate symbol p we mean a function d_p from $\{1,\dots,n\}$ to the set $\{+,-\}$. If $d_p(i)$ = '+' the i is called an *input* position of p and if $d_p(i)$ = '-' the i is called an *output* position of p. We write d_p in the form $p(d_p(1),\dots,d_p(n))$. Intuitively, queries formed by predicate p will be expected to have input positions occupied by ground terms. To simplify the notation, when writing an atom as $p(u,v)$, we assume that u is the sequence of terms filling in

A Prolog interpreter uses a particular ordering of branches, corresponding to top-down selection of rules of Π for resolution but the distinction is irrelevant since our results will hold for all such orderings.

the input positions of p and that v is the sequence of terms filling in the output positions. By $l(u,v)$ we denote expressions of the form $p(u,v)$ or *not* $p(u,v)$; $var(s)$ denotes the set of all variables occurring in s. Assignment of modes to the predicate symbols of a program Π is called *input-output specification.*

A rule $p_0(t_0, s_{m+1}) \leftarrow l_1(s_1,t_1), \ldots, l_m(s_m,t_m)$ is called *well-moded* w.r.t. its input output specification if for $i \in [1, m+1]$, $var(s_i) \subseteq \bigcup_{j=0}^{i-1} var(t_j)$. In other words, a rule is well-moded if

every variable occurring in an input position of a body goal occurs either in an input position of the head or in an output position of an earlier body goal;

ii) every variable occurring in an output position of the head occurs in an input position of the head, or in an output position of a body goal.

A program is called well-moded w.r.t. its input-output specification if all its rules are.

Apt and Pellegrini in [AP94] showed that if Π is well-moded (for some input output specification) and there is no rule in Π whose head contains more than one occurrence of the same variable in its output positions then Π is occur-check free w.r.t. any ground query q. Consider the following input output specification for Π_D.

$true\_after(-,+,+)$
$all\_true\_after(-,+,+)$
$false\_after(+,+,+)$
$one\_false\_after(+,+,+)$
$true\_at(-,+)$
$ab(+,+,+,+)$.
$causes(+,-,-)$
$contrary(-,-)$
$member(-,+)$
$imm\_follows(-,-)$
$occurs\_at(-,-)$

It is easy to check that Π_D is well-moded (w.r.t. this specification) and that it also satisfies the second condition of Apt-Pellegrini's Theorem. This concludes the proof. □

Lemma 7.6. For a simple domain description D, a query $\pi(q)$ to Π_D does not flounder, i.e., the LDNF-tree of $\pi(q)$ does not have nodes marked by flounder.

Proof. To prove the lemma we use another theorem from [AP94] which was also independently discovered by Stroetman [Str93]: if Π is well-moded (for some input output specification) and all predicate symbols occurring under *not* in Π are moded completely by input then a ground query $\pi(q)$ to Π does not flounder. The only two predicate symbols occurring under *not* in Π_D are one_false_after and ab. They both are completely input under the specification from Lemma 7.5. □

By proving Lemma 7.6, also Theorem 7.3 remains finally proved.

The next Theorems guarantee that the Prolog interpreter is complete w.r.t. Π_D^1, i.e., it does not miss any answers entailed by the declarative program and that its answers are sound w.r.t. D. We need the following lemma.

Lemma 7.7. For a simple domain description D, and for any query $\pi(q)$ to Π_D:

$$\Pi_D^2 \models \pi(q) \ iff \ \Pi_D^1 \models \pi(q)$$

PROOF. To prove the lemma we use the notion of *language tolerance* from [MT94].
Let Π be a program in a language $\mathcal{L}$, $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages containing $\mathcal{L}$, and Π_1 and Π_2 be instantiations of Π by ground terms of $\mathcal{L}_1$ and $\mathcal{L}_2$. Languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are called permissible for Π.
Program Π is called *language tolerant* if, for any two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ that are permissible for Π the following holds: for any consistent answer set A_1 of Π_1 there is a consistent answer set A_2 of Π_2 such that $A_1 \cap \mathcal{L}_2 = A_2 \cap \mathcal{L}_1$.
Obviously, $\Pi_D^2 \subset \Pi_D^1$, and therefore, the language $\mathcal{L}_1$ of Π_D^1 is an extension of the language $\mathcal{L}_2$ of Π_D^2 and both contain constants occurring in Π_D. To prove the lemma it suffices to show that Π_D is language tolerant. In [MT94] McCain and Turner give a sufficient condition for tolerance which uses the notions of stability and predicate-order-consistency. Let Π be a program with input output specification. A rule r:

$$p_0(s_0, t_0) \leftarrow p_1(s_1, t_1), \ldots, p_m(s_m, t_m), not\ p_{m+1}(s_{m+1}, t_{m+1}), \ldots, not\ p_n(s_n, t_n)$$

is said to be *stable* (w.r.t. this specification) if for any variable x from r, $x \in s_0$ or there is $1 \leq i \leq m$ such that $x \in var(t_i)$ and for any $1 \leq j < i$, $x \notin s_j$. Program Π is called stable w.r.t. an input output specification if all of its rules are. Π is stable if it is stable w.r.t. some input output specification (this definition of stability is given in [Str93]. McCain and Turner's definition is substantially more general). The notion of *predicate order consistency* from [MT94] is similar to the notion of order consistency from [Fag90]. Predicate dependency graph $G(\Pi)$ of a program Π has the nodes which are predicate symbols of Π. There is a positive edge in $G(P)$ from predicate symbol q to predicate symbol p if there is a rule in Π whose head contains expression $p(\ldots)$ and whose body contains expression $q(\ldots)$. There is a negative edge in $G(P)$ from predicate symbol q to predicate symbol p if there is a rule in Π whose head contains expression $p(\ldots)$ and whose body contains expression $not\ q(\ldots)$. We say that the relation $p \leq q$ holds iff there is a path in $G(\Pi)$ from p to q with an even number of negative edges and a path from p to q with an odd number of negative edges. Π is *predicate-order-consistent* if $p \leq q$ well-founded and there is no predicate symbol p in Π such that $p \leq p$. McCain and Turner prove that if Π is stable and predicate-order-consistent, then Π is language tolerant.
Given the definitions and the input-output specification from Lemma 7.5 it is easy to check that both conditions of the theorem are satisfied. This ends the proof of the lemma. □

Theorem 7.4. For any simple domain description D, its logic programming approximation Π_D, and a query q if $\Pi_D^1 \models \pi(q)$ then the Prolog's answer to $\pi(q)$ is yes, otherwise, the answer is no.
PROOF. Since, by Lemma 7.6 the LDNF tree for $\pi(q)$ does not have floundering nodes it suffices to prove that the Prolog interpreter always terminates, i.e.,

LDNF tree for $\pi(q)$ is finite. There are many sufficient conditions for termination in the literature. Probably the best known being the acyclicity condition from [AB90]. Unfortunately, because of the rules (e_1) and (SI), the program Π_D is not acyclic and hence we need a more subtle termination condition. We will use the notion of *acceptable* program from [AP91].
First, let us introduce some terminology. Let Π be a general logic program and p and q be predicate symbols from the language of Π. Then p *refers* to q if there is a rule Π with p in its head and q in its body, and p *depends on* q if $[p, q]$ belongs to the transitive closure of *refers*. Let Neg be the set of predicate symbol in Π that occur under *not* in the body of some rule in Π and let Neg* be the set of all predicate symbols in the language of Π on which the predicate symbols from Neg depend on. By Π^- we denote a program consisting of all rules from Π whose heads contain predicate symbols from Neg*. Its Clark's completion is denoted by $comp(\Pi^-)$. A model of Π is called *good* if its restriction to the predicates from Neg* is a model of $comp(\Pi^-)$. A mapping from ground atoms of Π into the set of natural numbers is called *level mapping* of Π. A program Π is *acceptable w.r.t. its level mapping* $|\ |$ *and interpretation* I if

   I is a good model of Π and
2. for every rule $A_0 \leftarrow L_1, \ldots L_m$ where L_i is an atom or an atom preceded by *not*, and any $0 < i < m$ we have that $I \not\models L_1, \ldots, L_i$ or $|A_0| > |L_{i+1}|$.

A program Π is called *acceptable* if it is acceptable w.r.t. some level mapping $|\ |$ and interpretation I of Π. Apt and Pedreschi in [AP91] proved that if Π is an acceptable program and *gamma* a ground query then all LDNF derivations of $\Pi \cup \gamma$ are finite and therefore the Prolog interpreter terminates on *gamma*.
To use this result we need to prove that Π_D is an acceptable program. First let us notice that $\Pi_D^- = \Pi_D$. As was shown before, Π_D^2 has stable model. This, together with Lemma 7.7, implies that Π_D^1 also has stable model, say I. It is well known that a stable model of a program is also a model of its Clark's completion which implies that I is a good model of Π_D.
Let c be the number of fluent literals in the language of D plus 1; p be a fluent literal or a list of fluent literals; r be an action or a list of action; and s_i $(0 \leq i \leq k)$ be a situation constant from D. Then

for any action a, $|a| = 1$, $|[\,]| = 1$, and for any list $[a|r]$ of actions $|[a|r]| = |r| + 1$.

Also, for any fluent literal f, $|f| = 1$, $|[\,]| = 1$, and for any list $[f|p]$ of fluent literals $|[f|p]| = |p| + 1$.

$|s_i| = i + 1$

Now we are ready to define a level mapping $|\ |$ for Π_D:

$|one\_false\_after(p, r, s)| = 4c * |s| + 2c * |r| + |p| + 3$

$|false\_after(f, r, s)| = 4c * |s| + 2c * |r| + |f| + 3$

$|all\_true\_after(p, r, s)| = 4c * |s| + 2c * |r| + |p| + 2$

$|true\_after(f, r, s)| = 4c * |s| + 2c * |r| + |f| + 2$

$|ab(f, a, r, s)| = 4c * |s| + 2c * (|r| + 1) + |f| + 1$

$|true\_at(f, s)| = 4c * |s| + 2c + |f| + 1$

while $|member|$ is equal to the length of its second parameter, $|contrary| = 1$, and all other atoms are mapped into 0.
It is not difficult to see that Π_D is acceptable w.r.t. $|\ |$ and I. This implies that every SLDNF tree for $\pi(q)$ is finite (including, of course, its LDNF tree) which completes the proof of the Theorem. □

Corollary 7.1. For any simple domain description D, its logic programming approximation Π_D, and a query q

$\Pi^1_D \models \pi(q)$ *iff the Prolog's answer to* $\pi(q)$ *is yes, and*

$\Pi^1_D \not\models \pi(q)$ *iff the Prolog's answer to* $\pi(q)$ *is no.*
PROOF. Follows immediately from theorems 3 and 4. □

The following theorem guarantees soundness of our Prolog based algorithm w.r.t. entailment in D.

Theorem 7.5. For any simple domain description D, its logic programming approximation Π_D, and a query q if the Prolog's answer to $\pi(q)$ is yes then $D \models q$.
PROOF. Notice that in Theorem 7.1 entailment w.r.t. Π_D corresponds to entailment in Π^2_D and hence we have that Π^2_D is sound w.r.t. D. Therefore, the proof follows immediately from Lemma 7.7 and Theorems 7.1 and 7.3. □

Corollary 7.2. For any simple domain description D with explicit complete information about the initial state, its logic programming approximation Π_D, and a query q

$D \models q$ *iff the Prolog's answer to* $\pi(q)$ *is yes, and*

$D \not\models q$ *iff the Prolog's answer to* $\pi(q)$ *is no.*
PROOF. Follows immediately from Theorem 7.2, Corollary 7.1 and Lemma 7.7.
□

8. IMPLEMENTING AN $\mathcal{L}$-PLANNER

In this section we use the logic programming approximation from Section 7 to construct an implementation of a $\mathcal{L}$-planner from Section 6. As before, we assume that the domain description D used by the planner is simple.

The implementation of $\mathcal{L}$-plan is obtained by combining Π_D (the logic programming approximation of the domain description D) with the rules cs_1—cs_4 that define the current situation, the rules $g_1 - - - g_2$ for breadth-first search through the space of possible action sequences[18], and the rule p that defines a generate-and-test planner.

$cs_1.\quad sit(s_0).$

$cs_2.\quad sit(S_{i+1}) \ :-\ sit(S_i),$
$\qquad\qquad\qquad\qquad imm\_follows(S_{i+1}, S_i).$

[18]We assume that relation action(X), read as "X is an action from domain D", is supplied by the programmer.

cs_3. $past(S)$:– $imm\_follows(S1, S)$.

cs_4. $current(S)$:– $sit(S)$,
$not\ past(S)$.

g_1. $generate([])$.

g_2. $generate([X|Y])$:– $generate(Y)$,
$action(X)$.

p. $find\_plan(\alpha, F)$:– $generate(\alpha)$,
$current(S)$,
$true\_after(F, \alpha, S)$.

Using the results from the previous section we can easily show that plans produced by the above planner are correct and that the shortest plan is found first. Moreover, Theorem 2 guarantees completeness of the program w.r.t. D for fully known initial situations, i.e., if there is a plan in D to achieve a given goal then the plan will be found. Of course, the program $\mathcal{L}$—plan does not always terminate, but we can limit the length of plans or guarantee termination by imposing some other conditions. The planner has several attractive features:

1. It is simple, has a clear declarative specification and can be proven correct w.r.t. this specification.
2. It allows (on-line) introduction of new objects into the domain, as well as addition of new information about values of fluents and occurrences of actions.
   The alphabet of $\mathcal{L}$ may have objects (fluents, actions, and situations) that do not appear in a particular domain description. Hence we may add new propositions to the domain description that contain objects that were not in the domain description before.
3. It is nonmonotonic, i.e., capable of creating new plans when new information invalidates the old one.
   Nonmonotonism is particularly useful in the design of autonomous agents [Mae91], where the agent may have to plan, execute part of the plan, observe the world and, if necessary, make a new plan.
4. It can easily incorporate heuristics and/or domain constraints (of both the static and dynamic kind).
   Domain constraints and heuristics may either be given as part of the specification or obtained via a learning program. For example, consider a case where we have domain knowledge that allows us to rule out the testing of certain sequences of actions. Now, if the domain has the property that if a sequence of actions α does not satisfy the domain constraints then no sequence of actions that has α as a prefix satisfies the domain constraints; then the rule g_2 can be modified into g'_2, defined as follows:
   $generate([X|Y])$:–$generate(Y), action(X), satisfies\_dom\_constr(X, Y)$.
   where $satisfies\_dom\_constr(X, Y)$ means that the sequence of actions $[X|Y]$ satisfies the domain constraints.
   As an example, the simple domain constraint saying that sequences of actions should not have the same action occurring consecutively can be expressed by the following rules together with g_1 and g'_2:

$satisfies\_dom\_constr(X, Y) :- not\ violates\_dom\_constr(X, Y).$
$violates\_dom\_constr(X, [X|Z]).$

5. It can be easily expanded to accommodate advances in action description languages.
Although planners are expanding their languages [BGPW93], currently most of them use a subset of ADL [Ped94]. Recently, research in logic programming theories of actions has progressed rapidly and we have logic programming theories that allow incomplete information about the initial state [GL92, DDS93, Dun93], concurrent actions [BG93, BT94], ramifications [Bar95], etc. Given a fluent f and a sequence of actions α, these theories determine the truth of f after execution of α. Hence, they can be directly used in conjunction with rules cs_1—cs_4, g_1—g_2 and p to extend $\mathcal{L}$-plan.

Before working on extending the planner to more general domains and doing serious comparison of our approach with other existing approaches to planning, we wanted to learn more about its efficiency. To do that we tested our planner on several domains[19] such as the blocks world, briefcase and the home-owner's domains [BGPW93, Wel94] that are used in evaluating planners.

When we expanded our planner with the additional knowledge that prohibits generation of impossible actions, the performance (execution time) of our planner was comparable with that of UCPOP[BGPW93].

We have also briefly looked at two other ways at improving performance:

a) to combine imperative and declarative paradigms of programming. For instance, we replaced the Prolog *generate* procedure by one written in C. This gave an order of magnitude improvement in performance without the loss of declarativness of Π_D that was used in testing;

b) to use multiple processors. The generate-and-test programs are the exact kind of programs that give maximum speed-up in a parallel computer with a parallel logic program interpreter/compiler (that exploits or-parallelism). Preliminary testing showed what we expected. The performance was close to n times better when we used n processors.

We would like to point out that our performance results are preliminary. They just indicate a possible trend. Since we did not test with a large number of randomly generated queries and the compiler used in our testing is different from that used for UCPOP[20], we do not in any way claim that our proposed planners are more efficient. However, we believe that $\mathcal{L}$—planners of the sort described in this section are very promising and deserve further investigation.

9. CONCLUSIONS

We have proposed the modification $\mathcal{L}_1$ of the action description language$\mathcal{A}$ capable of expressing actual situations, observations of the truth values of fluents in these

[19] These domains can be obtained from *ftp://june.cs.washington.edu/homes/weld/ucpop.html*.
[20] UCPOP is written in LISP while our programs are written in Quintus PROLOG or Quintus PROLOG + C.

situations, and observations of actual occurrences of actions. The entailment relation of $\mathcal{L}_1$ allows modeling of various types of hypothetical reasoning. This feature, together with the ability to denote the current actual situation, allows to reason about the design and correctness of plans in a changing environment.

We have discussed a possible application of $\mathcal{L}_1$ and its entailment relation to an architecture of intelligent agents and presented a provenly correct implementation of the planning module of this architecture in Prolog.

We plan to expand this work in several directions.

- The syntax and semantics of $\mathcal{L}_1$ should be expanded to deal with partially defined actions, to allow concurrent and non-deterministic actions [BG93, KL94, BT94], and global constraints [KL94], etc.
- More general and more efficient methods of computing the entailment of $\mathcal{L}_1$ and its extensions should be found. One possibility is to investigate other inference mechanisms which are sound w.r.t. declarative logic programming semantics. For programs with unique answer sets, the SLX procedure from [ADP94, ADP95] is a promising candidate. It computes answers w.r.t. the well-founded [VGRS91, PAA92] semantics of logic programs and hence is sound w.r.t. the answer set semantics. An interesting and challenging problem is to expand SLX by allowing disjunction and therefore reasoning by cases.
- Another promising direction of research is related to extension and elaboration of the planner from Section 8. One obvious approach is to modify the procedure *generate* to produce only those sequences of actions which are relevant to the goal and to use domain dependent heuristics.
- Finally, we also need to design and implement other modules in the architecture. Here the use of abduction seems to be promising.

Our work is a continuation of the approach of formalizing actions suggested in [GL92] which is deeply rooted in situation calculus [MH69, GLR91]. Our formalization, especially in its logic programming form, can be viewed as a combination of situation calculus with another prominent approach to formalizing actions – the event calculus of [KS86]. To the best of our knowledge the first paper combining the two in one formalism is [PR93]. Ideologically, their approach is similar to ours. In [PR93] situation calculus is presented as a theory of classical logic (with some second order features) and plays the role of our action description language. Our approach seems to allow more forms of incompleteness in the representation of the domain, but investigation of the precise relationship is a subject for future work. Some of the other recent works that combine event calculus and situation calculus are [MS94, Pro96, VBDDS95]. Amongst them the approach in [MS94] is closest to ours. Their function *state* that maps time points to situations is similar to our Σ which maps situation constants to sequences of actions. But they assume[21]that the domain description includes all occurrences of actions and they only allow fluent facts about the initial situations. Our approach is more general with respect to these restrictions. On the other hand, [MS94] contains discussions of allowing concurrent, divisible and overlapping actions, which we do not discuss in this paper.

[21]They do point out that these assumptions can be weakened using abduction.

Acnowledgements

We would like to acknowledge the grants NSF-IRI-92-11-662, NSF-IRI-95-01-577 and NSF-IRI-91-03-112. We would like to thank Vladimir Lifschitz, Norm McCain and Hudson Turner for useful discussions, Dan Weld for making available the UCPOP planner, Subbarao Kambhampati for pointing us to the planning literature. Special thanks to Rob Miller and anonymous referees for their comments which helped to significantly improve the quality of the paper. We would also like to thank Yulia Gelfond, Gopal Gupta and Enrico Pontelli for their help in running experiments.

REFERENCES

[AB90.] K. Apt and M. Bezem. Acyclic programs. In D. Warren and P. Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 617–633, 1990.

[AB91.] K. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3,4):335–365, 1991.

[AB94.] K. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19,20:9–71, 1994.

[AD94.] K. Apt and K. Doets. A new definition of SLDNF resolution. *Journal of Logic Programming*, 18:177–190, 1994.

[ADP94.] J. Alferes, C. Damasio, and L. Pereira. SLX — A top-down derivation procedure for programs with explicit negation. In *Proc. of International Logic Programming Symposium 94*, pages 424–438, 1994.

[ADP95.] J. Alferes, C. Damasio, and L Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning (special issue)*, 1995.

[ALP94.] J. Alferes, R. Li, and L. Pereira. Concurrent actions and changes in the situation calculus. In H. Geffner, editor, *Proc of IBERAMIA 94*, pages 93–104. McGraw Hill, 1994.

[AP91.] K. Apt and D. Pedreschi. Proving termination in general prolog programs. In *Proc. of the Int'l Conf. on Theoretical Aspects of Computer Software (LNCS 526)*, pages 265–289. Springer Verlag, 1991.

[AP94.] K. Apt and A. Pellegrini. On the occur-check free logic programs. *ACM Transaction on Programming Languages and Systems*, 16(3):687–726, 1994.

[Bar95.] C. Baral. Reasoning about Actions : Non-deterministic effects, Constraints and Qualification. In *Proc. of IJCAI 95*, pages 2017–2023, 1995.

[BG93.] C. Baral and M. Gelfond. Representing concurrent actions in extended logic programming. In *Proc. of 13th International Joint Conference on Artificial Intelligence,* pages 866–871, 1993.

[BGP96.] C. Baral, A. Gabaldon, and A. Provetti. Formalizing Narratives using nested circumscription. Proc. of the *14th AAAI National Conference on Artificial Intelligence*, 1996. An earlier version is in the Working notes of the third symposium *Logical Formalizations of Commonsense Reasoning,* pages 15–24.

[BGPW93.] A. Barrett, K. Golden, J. Penberthy, and D. Weld. UCPOP User's manual, version 2.0. Technical Report 93-09-06, Department of Computer Science and Engineering, University of Washington, 1993.

[BT94.] S. Bornscheuer and M. Thielscher. Representing concurrent actions and solving conflicts. In *Proc. of German Conference on AI*, 1994.

[Cla78.] K. Clark. Negation as failure. In Herve Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[CSW95.] W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–201, 1995.

[DDS93.] M. Denecker and D. De Schreye. Representing incomplete knowledge in abductive logic programming. In *Proc. of International Logic Programming Symposium 93*, pages 147–164, 1993.

[DM85.] P. Dembinski and J. Maluszynski. And-parallelism with intelligent backtracking for annotated logic programs. In V. Saraswat and K. Ueda, editors, *Proc of the International Symposium on Logic Programming*, pages 25–38, 1985.

[Dun93.] P. Dung. Representing actions in logic programming and its application in database updates. In D. S. Warren, editor, *Proc. of ICLP-93*, pages 222–238, 1993.

[Fag90.] F. Fages. Consistency of Clark's completion and existence of stable models. Technical Report 90-15, Ecole Normale Superieure, 1990.

[Geo94.] M. Georgeff, editor. *Journal of Logic and Computation, Special issue on Action and Processes*, volume 4 (5). Oxford University Press, October 1994.

[GL91.] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3,4), pages 365–387, 1991.

[GL92.] M. Gelfond and V. Lifschitz. Representing actions in extended logic programs. In *Joint International Conference and Symposium on Logic Programming.*, pages 559–573, 1992.

[GL95.] E. Giunchiglia and V. Lifschitz. Dependent fluents. In *Proc. of IJCAI 95*, pages 1964–1969, 1995.

[GLR91.] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In R. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe.* Kluwer Academic, Dordrecht, pages 167–180, 1991.

[HT93.] S Hölldobler and M Thielscher. Actions and specificity. In D. Miller, editor, *Proc. of ICLP-93*, pages 164–180, 1993.

[Kar93.] G. Kartha. Soundness and completeness theorems for three formalizations of action. In *IJCAI 93*, pages 724–729, 1993.

[Kar94.] G. Kartha. Two counterexamples related to Baker's approach to the frame problem. *Artificial Intelligence*, 69:379–391, 1994.

[KKY95.] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: a unified framework for evaluating design tradeoffs in partial order planning. *AI Journal (to appear), also in ASU-CSE-TR 94 002*, 1995.

[KL94.] G. Kartha and V. Lifschitz. Actions with indirect effects: Preliminary report. In *KR 94*, pages 341–350, 1994.

[KS86.] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[LMT93.] V. Lifschitz, N. McCain, and H. Turner. Automation of reasoning about action: a logic programming approach. In *Posters of the International Symposium on Logic Programming*, 1993.

[LS91.] F. Lin and Y. Shoham. Provably correct theories of actions: preliminary report. In *Proc. of AAAI–91*, 1991.

[LT94.] V. Lifschitz and H. Turner. Splitting a logic program. In P. Van Hentenryck, editor, *Proc. of the Eleventh Int'l Conf. on Logic Programming*, pages 23–38, 1994.

[Mae91.] P. Maes, editor. *Designing Autonomous Agents.* MIT/Elsevier, 1991.

[McC.] J. McCarthy. Overcoming an unexpected obstacle. Manuscript, 1992.

[McC63.] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford Artificial Intelligence Project: Memo 2, 1963.

[MH69.] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[MM82.] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM transaction on programming languages and systems*, 4:258–282, 1982.

[MS89.] W. Marek and V.S. Subrahmanian. The Relationship Between Stable, Supported, Default and Auto-Epistemic Semantics for General Logic Programs. In G. Levi and M. Martelli, editors, *Proceedings of Sixth International Conference in Logic Programming*, pages 600–620, 1989.

[MS94.] R. Miller and M. Shanahan. Narratives in the situation calculus. *Journal of Logic and Computation*, 4(5):513–530, 1994.

[MT94.] N. McCain and H. Turner. Language independence and language tolerance in logic programs. In *Proc. of the Eleventh Intl. Conference on Logic Programming*, pages 38–57, 1994.

[MT95.] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In *Proc. of IJCAI 95*, pages 1978–1984, 95.

[PAA92.] L. Pereira, J. Alferes, and J. Aparicio. Default theory for well founded semantics with explicit negation. In D. Pearce and G. Wagner, editors, *Logic in AI, Proc. of European Workshop JELIA'92 (LNAI, 633)*, pages 339–356, 1992.

[Ped88.] E. Pednault. Extending conventional planning techniques to handle actions with context-dependent effects. In *Proc. of AAAI-88*, pages 55–59, 1988.

[Ped94.] E. Pednault. ADL and the state-transition model of actions. *Journal of Logic and Computation*, 4(5):467–513, 1994.

[Pin94.] J. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, University of Toronto, Department of Computer Science, February 1994. KRR-TR-94-1.

[PR93.] J. Pinto and R. Reiter. Temporal reasoning in logic programming: A case for the situation calculus. In *Proceedings of 10th International Conference in Logic Programming*, pages 203–221, 1993.

[Pro96.] A. Provetti. Hypothetical reasoning about actions: from situation calculus to event calculus. *Computational Intelligence*, 12(3), 1996.

[Rei91.] 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*, pages 359–380. Academic Press, 1991.

[Str93.] K. Stroetman. A Completeness Result for SLDNF-Resolution. *Journal of Logic Programming*, 15:337–355, 1993.

[Tur93.] H. Turner. A monotonicity theorem for extended logic programs. In D. S. Warren, editor, *Proc. of 10th International Conference on Logic Programming*, pages 567–585, 1993.

[Tur94.] H. Turner. Signed logic programs. In *Proc. of the 1994 International Symposium on Logic Programming*, pages 61–75, 1994.

[VBDDS95.] K. Van Belleghem, M. Denecker, and D. De Schreye. Combining situation calculus and event calculus. In *Proc. of the twelfth international conference on Logic Programming*, pages 83–97, 1995.

[VGRS91.] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[Wel94.] D. Weld. An introduction to least commitment planning. *AI Magazine*,

15(4):27–61, winter 1994.
[Wor95.] Working notes of AAAI Spring Symposium. *Extending Theories of Action: Formal Theory and Practical Applications.* AAAI Press, 1995.

APPENDICES

HYPOTHESES IN DOMAIN DESCRIPTION

As promised in Section 5, we now consider the impact of allowing hypothesis (i.e., statements of the form (2)) in domain descriptions of $\mathcal{L}_1$. The definition of a model is now modified by adding the condition specifying when a hypothesis is true in an interpretation (see Definition 1) to Definition 3. All other definitions (in Section 3) remain the same. Let us now consider the following example.

Example .1. Consider a domain description

$$D_6 = \begin{cases} Facts: \\ (f_1) \quad \neg f_1 \wedge \neg f_2 \wedge \neg f_3 \textbf{ at } s_0 \\ (f_2) \quad s_0 \textbf{ precedes } s_1 \\ (f_3) \quad f_1 \textbf{ at } s_1 \\ Laws: \\ (l_1) \quad a_1 \textbf{ causes } f_1 \\ (l_2) \quad a_2 \textbf{ causes } f_1 \\ (l_3) \quad a_2 \textbf{ causes } f_2 \\ (l_4) \quad a_3 \textbf{ causes } f_2 \textbf{ if } f_1 \\ (l_5) \quad a_4 \textbf{ causes } f_3 \textbf{ if } f_2 \end{cases}$$

Consider the statement f_3 **after** a_4 **at** s_1 denoted by H. It is easy to see that D_6 has two models (Ψ, Σ_1) and (Ψ, Σ_2) where $\Sigma_1(s_1) = [a_1]$ and $\Sigma_2(s_1) = [a_2]$ and therefore

$$D_6 \models (H \supset a_2 \textbf{ occurs\_at } s_0)$$

Now, let $D_7 = D_6 \cup \{H\}$. D_7 has two models, (Ψ, Σ_2) and (Ψ, Σ_3) where $\Sigma_3(s_1) = [a_1, a_3]$ and hence does not entail a_2 **occurs_at** s_0.

Technically, this somewhat unexpected property is easy to explain. In the former case H has no influence on construction of models of D_6 and only plays a role in selecting models for which a_2 **occurs_at** s_0 must be evaluated. In the latter, instead, H is treated as a fact and influences the construction of the models of D_7.

Intuitively, the explanation can be found in the unclear ontological status of having (2) in a domain description. Because of the semantic ambiguities and following the general principle of simplicity in language design, we decided not to allow (2) in domain descriptions of $\mathcal{L}_1$.

LOGIC PROGRAMMING BACKGRUOUND

Let a general logic program is a collection of rules of the form

$$l_0 \leftarrow l_1, \ldots l_m, not\ l_{m+1}, \ldots, not\ l_n \quad (1)$$

where each l_i is a ground atom. The expression on the left hand (right hand) side of $\leftarrow$ is called the *head* (resp. the *body*) of the rule. Both the head and the body of (1) can be empty. Intuitively the rule can be read as: if $l_1, \ldots, l_m$ are believed and it is not true that $l_{m+1}, \ldots, l_n$ are believed then l_0 is believed. For a rule r of the form (1) the sets $\{l_0\}$, $\{l_1, \ldots, l_m\}$ and $\{l_{m+1}, \ldots, l_n\}$ are referred to as $head(r)$, $pos(r)$ and $neg(r)$ respectively. $lit(r)$ stands for $head(r) \cup pos(r) \cup neg(r)$. For any general logic program Π, $head(\Pi) = \bigcup_{r \in \Pi} head(r)$. For a set of predicates S, $Lit(S)$ denotes the set of all ground literals with predicates from S. For a general logic program Π, $Lit(\Pi)$ denotes the set of all ground literals with predicates from the language of Π. When it is clear from the context we write Lit instead of $Lit(\Pi)$.

The *answer set* (stable model) of a general logic program Π without negation-as-failure is the smallest subset S of atoms such that for any rule $l_0 \leftarrow l_1, \ldots, l_m$ from Π, if $l_1, \ldots, l_m \in S$, then $l_0 \in S$. The answer set of a program Π that does not contain negation-as-failure is denoted by $\alpha(\Pi)$.

Now let Π be *any* general logic program. For any set $S \subset Lit$, let Π^S be the general logic program obtained from Π by deleting

(i) each rule that has a formula $not\ l$ in its body with $l \in S$, and
(ii) all formulae of the form $not\ l$ in the bodies of the remaining rules.

Clearly, Π^S does not contain not, so that its answer set is already defined. If this answer set coincides with S, then we say that S is an *answer set* of Π. In other words, the answer sets of Π are characterized by the equation $S = \alpha(\Pi^S)$.

For a general program Π and an atom f we say $\Pi \models f$ iff f is $true$ in all answer sets (stable models) of Π. Similarly, we say $\Pi \models \neg f$ iff f is $false$ in all stable models of Π.

Lemma .1 (Marek and Subrahmanian [MS89]). For any answer set S of a general logic program Π:

For any rule of type 1 from Π, if

$$\{l_1, \ldots, l_m\} \subseteq S \text{ and } \{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$$

then $l_0 \in S$;

ba)) *If S is a consistent answer set of Π and $l_0 \in S$ then there exists a rule of the type 1 from Π such that*

$$\{l_1, \ldots, l_m\} \subseteq S \text{ and } \{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$$

□

We now review the definitions of splitting which is used in the proof of the lemmas.

Definition .1. [[LT94]](Splitting set) A *splitting set* for a program Π is any set U of literals such that, for every rule $r \in \Pi$, if $head(r) \cap U \neq \emptyset$ then $lit(r) \subset U$. If

U is a splitting set for Π, we also say that U splits P. The set of rules $r \in \Pi$ such that $lit(r) \subset U$ is called the *bottom* of Π relative to the splitting set U and denoted by $b_U(\Pi)$. The subprogram $\Pi \setminus b_U(\Pi)$ is called *the top of* Π relative to U. □

Definition .2. [[LT94]] (Partial evaluation) The partial evaluation of a program Π with splitting set U w.r.t. a set of literals X is the program $e_U(\Pi, X)$ defined as follows. For each rule $r \in \Pi$ such that:

$$(pos(r) \cap U) \subset X \quad \wedge \; (neg(r) \cap U) \cap X = \emptyset$$

put in $e_U(\Pi, X)$ the rule $r^{'}$ which satisfies the following property:

$$head(r') = head(r), \; pos(r') = pos(r) \setminus U, \; neg(r') = neg(r) \setminus U$$

Definition .3. (Solution) [LT94] Let U be a splitting set for a program Π. A solution to Π w.r.t. U is a pair $\langle X, Y \rangle$ of sets of literals such that:

1. X is an answer set for $b_U(\Pi)$;
2. Y is an answer set for $e_U(\Pi \setminus b_U(\Pi), X)$ and
3. $X \cup Y$ is consistent.

Lemma .2 ([LT94]). (Splitting Lemma) Let U be a splitting set for a program Π. A set A of literals is a consistent answer set for Π if and only if $A = X \cup Y$ for some solution $\langle X, Y \rangle$ to Π w.r.t. U. □