CS2K707(P) Seminar Report

on

A LOGIC PROGRAMMING APPROACH TO KNOWLEDGE STATE PLANNING

Submitted In Partial Fulfilment Of The Degree Of

Bachelor Of Technology

by

K.Gajaruban
Y1.301,S7 CSE

Department of Computer Science & Engineering
National Institute of Technology, Calicut
2004 Monsoon

National Institute of Technology, Calicut
Department of Computer Science & Engineering

Certified that this Seminar Report entitled

A LOGIC PROGRAMMING APPROACH TO KNOWLEDGE STATE PLANNING

is a bonafide report of the Seminar presented by

K.Gajaruban
Y1.301,S7 CSE

in partial fulfilment of the degree of
Bachelor of Technology

Vinod 16 Sept '04

Mr.Vinod P.
Seminar Coordinator
Lecturer
Dept.of Computer Science & Engineering

Dr.V.K.Govindan
Professor and
Head
Dept.of Computer Science & Engineering

Abstract

It is a proposal of new deaclarative planning language namely K, which conforms to logic programming paradigm. This language is capable of describing the transitions between different knowledge states rather transitions between completely described states of the world. Therefore this language allows planning under incomplete knowledge. Still it supports default principles of logic programming like negation as failure. At the same time transitions between completely described states of world is also possible. So it's very flexible. This allows natural and compact problem representation. A thorough analysis of computational complexity of K is given in the report, using examples of planning problems. It includes secure planning and standard planning. Under various restrictions these complexities range from NP to NEXPTIME. This forms a theoritical basis for the implementations of K on top of the DLV system, resulting in DLVk.

Contents

1 Introduction to knowledge state planning in a historical persective

Planning is an important aspect of AI. So planning capability has always been a problem from the begining itself. A numerous methods have been studied and developed over the decades. It started with the McCarthy's proposal in 1950s. As a breakthrough Robinson's resolution method laid the basis for deductive planning and the wellknown situaion calculus (McCarthy and Hayes in 1969). But due to some problems like frame problem it lost the popularity, while STRIPS approach which is a hybrid between logic and procedural computation, gained the importance. No other logic base planning system was evolved for a long time. But then in the last 15 years logic base planning celebrated a renaissance as a result of the following.

- Solution to the frame problem was found. So the situation calculus led GOLOG planning language by Toronto group.
- Planning problems were formulated as logical satisfiable problems.(Kautz and Selmen in 1992). It was possible to solve large problems which can't be solved even by specialized planning system. It led to the efficient blackbox planning system. At the same time planning problems were reduced to be solved using logic programming, model checking and boolean formulas.
- Planning was brought as a task in logic base languages for reasoning about actions. It led to the causal calculator(CCALC) and the C plan system which is based on important C action language.

Answer set programming was proposed as a tool for problem solving. In this, planning problems are formulated in a domain independant planning language and are mapped to logic program such that answer set of the program give the solution to the planning problem. Planners can be created this way to support expressive action descriptive languages by the use of efficient answer set engines like Smodel or DLV. This suggestion is pursued to develop this K language. It's named K to emphasize that this language describes states of knowledge where as the other languages like C were based on classical logic and described the states of world. A state of the world is characterize by the truth values of fluents. That is in a predicate describing the relevant properties of domain of discourse, fluents have to take either true or false. Accordingly an action is taken if the precondition is true in the current state. But in reality, planning agents don't have a complete knowledge. So number of fluents are unknown. But a decision has to be taken based on this. To overcome this this language K adopts a three valued view of fluents, in which their values can be true or false or unknown. So planning based on complete knowledge is taken as a special case. It is closer in spirit to answer set semantics whereas the other planning languages stick to classical logic. This is useful when dealing with incomplete knowledge. K doesn't adopt possible world's view of knowledge states and reason about possible cases for determining knowledge state transitions. Futhermore computaional complexity of K is analysed, which provides a platform for theoritical DVLk implementations. It is a powerful deaclarative planning system.

2 Language K

Syntax and semantics of the laguage K is discussed in detail here.

2.1 Basic syntax

2.1.1 Actions, Fluents and Types

Let E(act),E(fl) and E(typ) be disjoint sets of action, fluent and type respectively. These names are effectively predicate symbols associated with arity(>= 0). Here E(act) and E(fl) are used to describe dynamic knowledge and E(typ) is used to describe static background knowledge. We tacitly assume E(typ) contains built in predicates in particular equality ('=') which are not shown. Furthermore let E(con) and E(var) be disjoint sets of constant and variable symbols respectively.

2.1.2 Definition 1

Given P is an element of E(act) (respectively E(fl), E(typ)), an action (respectively fluent, type) atom is defined as p(t1.....tn) where n is the arity of p and t1,.....tn are all elements of E(con) or E(var). An action (respectively fluent, type) literal is an action (respectively fluent, type) atom a or it's negation -a where - is the negation symbol.An usual a literal is ground if it doesn't contain any variable. Given a literal l let -l denote it's complement, that is -l=a if l=-a and -l=-a if l=a, where a is an atom. A set L of literals is consistant if L intersection -L is an empty set. Further L+ and L- indicate set of positive and negative literals in L respectively. Set of all action literals is denoted by L(act). Same way for fluent and type as well. Further L(act,fl) is the union of L(act) and L(fl). And L(dyn) stands for dynamic literals. L(dyn) L(fl) union L(act)+.

2.1.3 Definition 2

Action and fluent declarations are of these forms. p(X1,.....Xn) requires t1,....tn. Where p is the element of L(act)+ or L(fl)+ accordingly. X1,....Xn are elements of E(var). Where $n >= 0$ is the arity of p. t1,....tm are elements of L(typ), $m >= 0$, and every Xi occures in t1,...,tm. If m=0 the keyword requires may be ommited. In the following we generically refer to action and fluent declarations as type declarations when no further distinction is necessary. Next comes the definition of causation rules, by which static and dynamic dependencies of one fluents on other fluents and actions are specified.

2.1.4 Definition 3

Causation rule is an expression of the form caused f if b1,...,bk, not bk+1,...,not bl after a1,...,am, not am+1,...,not an. where f is an element of L(ft) union false. b1,...,bl are elements of L(fl,typ), a1,...an are elements of L. l>= $k >= 0$ and n>= $m >= 0$. Rules when n=0 are referred to as static rules and all others are dynamic rules. When l=0 the keyword if is omitted. Likewise when n=0 the keyword after is omitted. If l=n=0 then caused is optional. Ti access the parts of the causation rule r the following notations are used. h(r)={f},post+(r)={b1,...,bk},post-(r)={bk+1,...,bl},pre+(r)={a1,...,am},pre-(r)={am+1,...,an}, and lit(r)={f,b1,...,bl,a1,...,an}. Intuitively pre+(r) accesses the state before some actions happen, and post+(r) the part after the action has been executed. While the scope of general static rules is over all knowledge states, it is often useful to specify rules only for the initial states.

2.1.5 Definition 4

An initial state constraint is a static rule of the form as mentioned in the previous definition preceeded by keyword initially. K also allows STRIPS like conditional execution of actions, where k allows several alternative executability conditions for an action. This is beyond the standard STRIPS notion.

2.1.6 Definition 5

An executability condition is an expression of the form executable a if b1,...,bk not bk+1,...,not bl where a is an element of L(act)+ and b1,...,bl are the elements of L and l>= k >= 0. If l=0 keyword 'if' is skipped. Given an executability condition e, we access its parts with h(e) = {a}, pre+(e)=1,...,bk}, pre-(e)={bk+1,...,bl}, and lit(e) = {a,b1,...,bl}. Intuitively pre-(e) refers to the state at which some action's suitability is evaluated. here as opposed to causation rules a state after the exexcution os actions is not considered. So no part of post−(r) is needed. Neverthless post+(e)=post-(e)=NULL is defined for conveviencc. For any executability condition initial state constraint r is defined as post(r)= post+(r) union post-(r), pre(r)=pre+(r) union pre-(r), and b(r) = b+(r) union b-(r), where b+(r) = post+(r) union pre+(r), and b-(r) = post-(r) union pre-(r). Consider an example, where E(typ) ={r,s}, E(fl) = {f}, and E(act) = {ac}:

- d1: f(X) requires -r(X,Y), s(Y,Y).
- d2: ac(X.Y) requires s(X.Y).
- r1: caused f(X) if s(X,X), not -f(X) afte ac(X,Y), not -r(X,X).
- e1: executable ac(X,Y) if s(Z,Y), not f(X), Z<> Y.

Then we have h(r1)={f(X)}, pre(r1)={ac(X,Y), -r(X,X)} and post(r1)={s(X,X), -f(X)}. Further h(e1)=ac(X,Y) and pre(e1) = {s(Z,Y), f(X), Z <> Y}; Safety restriction all rules have to satisfy some sysnctactic restrictions, which is similar to notion of safety in logic programs. All variables in a default negated type literal must also occur in some literal which is not a default negated type literal. Thus safety is required only for variables appearing in default negated type literals, while it is not required at all for variables appearing in fluent and action literals. The reason is that the range of the latter variables appearing in fluent and action literals. The reason is that the range of the latter variables is implicitly restricted by the respective type declarations. Observe that the rules in the previous examples are safe.

Planning domain and planning problems Consider any pair (D,R) where D is s finite set of action and fluent declarations and R is a finite set of safe causation rules, safe initialstate constraints, and safe executability conditions, an action description.

2.1.7 Definition 6

A planning domain is a pair PD = (n,AD), where n is a Datalog program over the literals of L(typ) (background knowledge)which is assumed to be safe in a standard LP sense and to have a total well founded model, and AD is an action description. We say that PD is positive. if no default negation occurs in AD. If program n has a total well founded model M. then M is the unique answer set of n. In particular each stratified program n has a total well founded model. The semantic condition of a total well founded model admits a limited use of unstratified negation, which is convenient for knowledge representation purposes, and in particular for expressing default properties. Planning domains represent the universe of discourse for solving concrete planning problems, which are defined next.

2.1.8 Definition 7

A planning problem p=(PD,q) is a pair of planning domain PD and a query q, where a query ia an expression of the form

g1,...,gm, not gm+ 1,...,not gn ? (i)

where g1,....,gn are elements of L(fl) are variable free, n>= m >= 0, and i >= 0 denotes the plan length.

2.2 Semantics

For defining the semantics of K planning domains and planning problems, we start with the preliminary definition of the typed instantiation of a planning domain. This is similar to the grounding of a logic program, with the difference being that only correctly typed fluent and action literals are generated.

2.2.1 Typed instantiation

Let substitutions and their application to syntactic objects be defined as usual (i.e., assignments of constants to variables that replace the variables throughout the objects).

2.2.2 Definition 8

Let PD = (n, (D,R)) be a planning domain, and let M be the (unique) answer set of n [Gelfond and Lifschitz 1991]. Then, O(p(X1, . . . , Xn)) is a legal action (respectively, fluent) instance of an action (respectively, fluent) declaration d element of D of the form (1), if O is a substitution defined over X1, . . . , Xn such that {O(t1), . . . , O(tm)} ? M. By Lpd, we denote the set of all legal action instances, legal fluent instances (also referred to as positive legal fluent instances) and classically negated () legal fluent instances (negative legal fluent instances). Based on this, we now define the instantiation of a planning domain respecting type information as follows.

2.2.3 Definition 9

For any planning domain PD = (n, (D,R)) its typed instantiation is given by PD(down)=(n(down), (D,R(down)) where n(down) is the grounding of n (over E(con)) and R(down) = {O(r) — r element of R , O element of @ }, where @ is the set of all substitutions O of the variables in r using E(con), such that lit(O(r)) intersection L(dyn) is a proper subset of Lpd. In other words, in PD(down) we replace n and R by their ground versions, but keep of the latter only rules where the atoms of all fluent and action literals agree with their declarations. We say that a PD = (n, (D,R) is ground, if n and R are ground, and moreover that it is well typed, if PD and PD(down) coincide.

2.2.4 Definition 10

A state with respect to a planning domain PD is any consistent set s proper subset of L(fl) intersection Lpd of positive and negative legal fluent instances. A tuple t = (s,A,s') where s, s' are states and A a proper subset of L(act) intersection Lpd is a set of legal action instances in PD is called a state transition. Observe that a state does not necessarily contain either f or f for each legal instance f of a fluent. In fact, a state may even be empty (s = NULL). The empty state represents a .tabula rasa. state of knowledge about the fluent values in the planning domain. Furthermore, in this definition, state transitions are not constrained.this will be done in the definition of legal state transitions, which we develop now. To ease the intelligibility of the semantics, we proceed in two steps. Let us first define the semantics for positive planning problems, i.e., planning problems without default negation, and then we define the semantics of general planning domains by a reduction to positive planning domains. In what follows, we assume that PD = (n, (D,R)) is a ground planning domain that is well typed, and that M is the unique answer set of n. For any other PD, the respective concepts are defined through its typed grounding PD(down).

2.2.5 Definition 11

A state s0 is a legal initial state for a positive PD, if s0 is the smallest (under inclusion) set such that post(c) a proper subset of s0 union M implies h(c) is also a proper subsrt of s0, for all initial state constraints and static rules c is an elemet of R . For a positive PD and a state s, a set A is an element of L(act)+ is called executable action set with respect to s, if for each a element of A there exists an executability condition e element of R such that h(e) = {a}, pre(e) intersection L(fl,typ) a proper subset of s union M, and pre(e) intersection L(act)+ a proper subset of A. Note that this definition allows for modeling dependent actions, that is, actions that depend on the execution of other actions.

2.2.6 Definition 12

Given a positive PD, a causation rule r element of R is satisfied by a state s' with respect to a state transition t = (s,A,s') if and only if either h(r) a proper subset of s'
false} or not all of (i).(iii) hold: (i) post(r) a proper subset of s' union M, (ii) pre(r) intersection L(fl,typ) a proper subset of s union M, and (iii) pre(r) intersection L(act) is a proper subset of A. A state transition t = (s,A,s') is called legal, if A is an executable action set with respect to s and s' is the minimal consistent set that satisfies all causation rules in R except initial state constraints with respect to t. The above definitions are now generalized to a well typed ground PD containing default negation by means of a reduction to a positive planning domain, which is similar in spirit to the Gelfond.Lifschitz reduction [1991].

2.2.7 Definition 13

Let PD be a ground and well-typed planning domain, and let t = (s,A,s') be a state transition. Then, the reduction PDt= (n, (D,Rt)) of PD by t is the planning domain where Rt is obtained from R by deleting (1) every causal rule, executability condition, and initial state constraint r element of R for which either post-(r) intersection (s' union M) != NULL or pre-(r) intersection (s union A union M) != NULL holds, and (2) all default literals not L (L element of L) from the remaining r element of R . Note that PDt is positive and ground. Legal initial states, executable action sets, and legal state transitions are now defined as follows.

2.2.8 Definition 14

Let PD be any planning domain. Then, a state s0 is a legal initial state, if s0 is a legal initial state for PDt , where t = (NULL, NULL, s0); A set A is an executable action set in PD with respect to a state s, if A is executable with respect to s in PDt with t = (s,A,NULL); and, a state transition t = (s,A,s') is legal in PD, if it is legal in PDt .

2.2.9 Example

Reconsider the type declarations d1 and d2, causation rule r1 and executability condition e1 in the previous example. Suppose E(con) contains two constants a and b, and that the background knowledge = has the following answer set: M = {-r(a, b), r(b, a), s(a, a), s(a, b), s(b, b)}. Then, for example, f(a) is a legal fluent instance of d1, f(X) requires - r(X, Y), s(Y, Y). where

O = {X = a, Y = b}. Similarly, ac(a, b) is a legal action instance of declaration d2, ac(X, Y) requires s(X, Y). where O = {X = a, Y = b}. Thus, f(a) and ac(a, b) belong to Lpd. The empty set s0 = AE is a legal initial state, and in fact the only one since there are no initial state constraints or static causation rules in PD, and thus also not in PDt for every t = (NULL, NULL, s0). The action set A = {ac(a, b)} is executable with respect to s0, since for t = (s0,A,NULL), the reduct PDt contains the executability condition e=1 : executable ac(a, b) if s(a, b), $a <> b$. and both s(a, b) and $a <> b$ are contained in s0 union M. Thus, we can easily verify that t = (s0,A,s1), where A = {ac(a, b)} and s1 = {f(a)} is a legal state transition:

PDt contains a single causation rule
r=1 : caused f(a) if s(a, a) after ac(a, b).
which results from r1 for O = {X = a, Y = b}. Clearly, s1 satisfies this rule, as h(r1') is a proper subset of s1, and s1 is smallest, since s(a, a) is an element of M and ac(a, b) element of A holds. On the other hand, t = (s0,A',s1), where A' = {ac(a, b), ac(b, b)} is not a legal transition: while ac(b, b) is a legal action instance, there is no executability condition for it in PD(down)t, and thus ac(b, b) is not executable in PD with respect to s0.

2.2.10 Plans

After having defined state transitions, we now formalize plans as suitable sequences of states transitions which lead from an initial state to some success state which satisfies a given goal.

2.2.11 Definition 15

A sequence of state transitions T = ((s0,A1,s1), (s1,A2,s2), . . . , (sn-1,An,sn), n >= 0. is a trajectory for PD, if s0 is a legal initial state of PD and all (si-1,Ai,si), 1 <= i <= n, are legal state transitions of PD. Note that in particular, T = () is empty if n = 0.

2.2.12 Definition 16

Given a planning problem p = (PD, q), where q has form (4), a sequence of action sets (A1, . . . , Ai), i >= 0, is an optimistic plan for P, if a trajectory T = ((s0,A1,s1), (s1, A2, s2), . . . , (si-1, Ai , si)) in PD exists such that T establishes the goal, that is, {g1, . . . , gm} a proper subset of si and {gm+1, . . . , gn} intersection si = NULL. The notion of optimistic plan amounts to what in the literature is defined as .plan. or .valid plan. etc. The term .optimistic. should stress the credulous view underlying this definition, with respect to planning domains that provide only incomplete information about the initial state of affairs and/or bear nondeterminism in the action effects, that is, alternative state transitions. In such domains, the execution of an optimistic plan P is not a guarantee that the goal will be reached. We therefore resort to secure plans (alias conformant plans), which are defined as follows.

2.2.13 Definition 17

An optimistic plan (A1, . . . , An) is a secure plan, if for every legal initial state s0 and trajectory T = ((s0, A1, s1), . . . , (sj-1, Aj , sj)) such that 0 <= j <= n, it holds that (i) if j = n then T establishes the goal, and (ii) if $j < n$, then Aj+1 is executable in sj with respect to PD, that is, some legal transition (sj , Aj+1, sj+1) exists. Observe that plans admit in general the concurrent execution of actions at the same time. However, in many cases, the concurrent execution of actions may not be desired (and explicitly prohibited, as discussed below), and attention focused to plans with one action at a time. More formally, we call a plan (A1, . . . , An) sequential (or nonconcurrent), if mod of Aj >= 1. for all 1 <= j <= n.

3 Knowledge representation in K

In this section, the use of K for modeling planning problems is explored by examples. Special attention is given to features and techniques which distinguish K from similar languages.

3.1 Deterministic Planning with Complete Initial Knowledge

We first study a simple setting in which the planning domain is not subject to nondeterminism and the planning agent has complete knowledge of the initial state of affairs. For later reference, we formally introduce the following notion.

3.1.1 Definition 1

Let PD be a planning domain. Then, a legal transition (s, A, s1) in PD is determined, if s1 = s2 holds for every possible legal transition (s, A, s2) (i.e., executing A on s leads to a unique new state). We call PD deterministic, if all legal transitions in it are determined. Consider first the planning problem depicted in Figure 1, which is set in the blocksworld. This problem illustrates the famous Sussman anomaly [Sussman 1990]. We will first describe the planning domain PDbwd = (nbw, (Dbwd , Rbwd)) of blocksworld. It involves distinguishable blocks and a table. Blocks and the table can serve as locations on which other blocks can be put (a block can hold at most one other block, while the table can hold arbitrarily many blocks). We thus define the notions of block and location in the background knowledge nbw as follows:
block(a). block(b). block(c).
location(table).
location(B) : - block(B).
For representing states, we declare two fluents in FDbwd : on states the fact that some block resides on some location, occupied is true for a location, if its capacity of holding blocks is exhausted.

fluents : on(B, L) requires block(B), location(L). occupied(B) requires location(B).
Only one action is declared in ADbwd : move represents moving a block to some

location (implicitly removing it from its previous location).
actions : move(B, L) requires block(B), location(L).

Let us now specify the initial state constraints IRbwd . For the initial state, occupied does not have to be specified, as it follows from knowledge about on. Note that only positive facts are stated for on, nevertheless the initial state is unique because the fluent on is interpreted under the closed world assumption (CWA) [Reiter 1978], that is, if on(B, L) does not hold, we assume that it is false.
initially : on(a, table). on(b, table). on(c, a).
Next, we specify causation rules and executability conditions CRbwd . First a static rule is given, defining occupied for blocks if some other block is on them.
always : caused occupied(B) if on(B1, B), block(B).
A move action is executable if the block to be moved and the target location are distinct (a block cannot be moved onto itself). A move is not executable if either the block or the target location is occupied.

executable move(B, L) if B <> L.
nonexecutable move(B, L) if occupied(B).
nonexecutable move(B, L) if occupied(L).

The action effects are defined by dynamic rules. They state that a moved block is on the target location after the move, and that a block is not on the location on which it resided before it was moved.

caused on(B, L) after move(B, L).
caused - on(B, L1) after move(B, L), on(B, L1), L <> $L1$.
Next we state that the fluent on should stay true, unless it becomes false explicitly. Note that we need not specify this property for occupied, as it follows from on via the static rule.
inertial on(B, L).

It is worthwhile noting that in this example the fluents are represented positively and their negation is usually implicit via the closed world assumption. Therefore, for example we do not need to declare -on(B, L) as inertial. There is one exception in a rule describing a negative action effect: Here, the negation becomes known explicitly, and its purpose is the termination of the inertial truth of an instance of on. However, we do not need to remember this negative knowledge by inertia. In this sense, K allows to formalize .forgetting. about information, such that we can keep only the .necessary. information in the domain of discourse.

In order to solve the original planning problem, we associate the following goal qbwd for plan length 3 to PDbwd , yielding Pbwd :
goal : on(c, b), on(b, a), on(a, table) ? (3)
Pbwd allows a single sequential plan of length 3:
({move(c, table)}, {move(b, a)}, {move(c, b)})
Thus, the above plan requires to first move c on the table, then to move b on top of a, and finally, to move c on b. It is easy to see that this sequence of actions leads to the desired goal. Since this domain is deterministic and has a unique initial state, all optimistic plans are also secure. We remark that the above representation is tailored for sequential planning, since the executability conditions do not take possible parallel moves properly into account. For example, moving the same object to different locations would have to be excluded, if parallel moves were allowed.

3.2 Planning with Incomplete Initial State Descriptions

In the example of Section 3.1, it is assumed that the initial state is correct (with respect to the domain in question) and fully specified (thus unique). In this section we explore how these implicit requirements can be weakened. As an accompanying example problem, suppose that there is a further block d in the original planning problem of blocks mentioned first The exact location if d is unknown, but we know that it is not on top of c. Furthermore, there is a slightly different goal involving d. The problem is depicted in second example. We will define a corresponding planning domain PDbwi = ((nbwi , (Dbwi , Rbwi)) by extending PDbwd . The additional knowledge about the initial state is represented by adding -on(d, c). to IRbwi . and the background knowledge nbwi is obviously enriched by the fact block(d).

Let us first consider the necessary extensions for handling cases in which the initial state description cannot be assumed to be correct (e.g., when completing the partial initial state description, incorrect initial states can arise). The following conditions should be verified for each block:

- (i) It is on top of a unique location,
- (ii) it does not have more than one block on top of it, and
- (iii) it is supported by the table (i.e., it is either on the table or on a stack of blocks that is on the table) [Lifschitz 1999b].

It is straightforward to formulate conditions (i) and (ii) and include them into IRbwi : initially : forbidden on(B, L), on(B, L1), L <> $L1$. forbidden on(B1, B), on(B2, B), block(B). B1 <> $B2$. For condition (iii), we add a fluent supported to FDbwi , which should be true for any block in a legal initial state: fluents : supported(B) requires block(B). We add the definition of supported and a constraint stating that each block must be supported to IRbwi .
initially : caused supported(B) if on(B, table).
caused supported(B) if on(B, B1), supported(B1).
forbidden not supported(B).
Any planning problem involving the domain defined so far does not admit any plan if the initial state is either incorrectly specified or incomplete in the sense that not all block locations are known (as supported will not hold for these blocks). Note that the action move preserves the properties (i), (ii), (iii) above for sequential plans; it is therefore not necessary to check these properties in all states if concurrent actions are not allowed. Next we show how incomplete initial states can be completed in K. To this end, we use the keyword total (defined before), and simply add total on(X, Y). to IRbwi . In this way, all possible completions with respect to on(X, Y) serve as candidate initial states, only some of which satisfy the initial state constraints, making them legal initial states. For example, the state in which on(d, a) holds is not legal as the constraint which checks condition (ii) is violated. Finally, let us consider the planning problem Pbwi = (PDbwi , qbwi), where qbwi is
goal : on(a, c), on(c, d), on(d, b), on(b, table) ? (j)
Usually, when dealing with incomplete knowledge, we look for plans which establish the goal for any legal initial state (in this particular case case no matter whether on(d, b) or on(d, table) holds), so we are interested in secure plans. The following secure sequential plan exists for Pbwi and j = 4: ({move(d, table)}, {move(d, b)}, {move(c, d)}, {move(a, c)}).

It is easily verifiable that this plan works on each legal initial state: Since d is not occupied in any legal initial state, the first action can always be executed. In some cases, one is interested in a plan that works for some possible initial state: For Pbwi , an optimistic plan exists for j = 2:
({move(c, d)}, {move(a, c)}).
It works only for the initial state in which on(d, b) holds, and fails for all other admissible initial states. Hence, it is not a secure plan.

4 Planning complexity

The results on the complexity of planning in K are related to several results in the planning literature. First and foremost, planning in STRIPS can be easily emulated in K planning domains, and thus results for STRIPS planning carry over to respective planning problems in K, in particular Optimistic Planning, which by the results in Bylander [1994] and Erol et al. [2000] is PSPACEcomplete. As for finding secure plans (alias conformant or valid plans), there have been interesting results in the recent literature. Turner [2002] has analyzed in a recent paper the effect of various assumptions on different planning problems, including conformant planning and conditional planning under domain representation based on classical propositional logic. In particular, Turner reports that deciding the existence of a classical (i.e., optimistic) plan of polynomial length is NP-complete, and NP-hard already for length 1 where actions are always executable. Furthermore, he reports that deciding the existence of a conformant (i.e., secure) plan of polynomial length is E P3 complete, and E P3 hard already for length 1. Furthermore, the problem is reported E P2 complete if, in this terminology, the planning domain is proper, and E P2 hard for length 1 in deterministic planning domains. Turner's results match our complexity results, announced in Eiter et al. [2000]; This is intuitively sound, since answer set semantics and classical logic, which underlies ours and his framework, respectively, have the same computational complexity. Giunchiglia [2000] considered conformant planning in the action language C, where concurrent actions, constraints on the action effects, and nondeterminism on both the initial state and effects of the actions are allowed. All these features are provided in our language K as well. Furthermore, Giunchiglia presented the planning system C-plan, which is based on SAT solvers for computing, in our terminology, optimistic and secure plans following a two-step approach. For this purpose, transformations of finding optimistic plans and security checking into SAT instances and QBFs are provided. The same approach is studied in Ferraris and Giunchiglia [2000] for an extension of STRIPS in which part of the action effects may be nondeterministic. While not explicitly analyzed, the structures of the QBFs emerging in Giunchiglia [2000] and Ferraris and Giunchiglia [2000] reflect our complexity results for Optimistic Planning and Security Checking

Rintanen [1999a] considered planning in a STRIPS style framework. He showed that, in our terminology, deciding the existence of a polynomial length sequential optimistic plan for every totalization of the initial state, given that actions are deterministic, is E P2 complete. Furthermore, Rintanen showed how to extract a single such plan P which works for all these totalizations, from an assignment to the variables X witnessing the truth of a QBF *X"YZ* @ that is constructed in polynomial time from the planning instance. Thus, the associated problem of deciding whether such a plan P exists is in E P3 . Note that intuitively, checking suitability of a given optimistic plan is in this problem more difficult than Security Checking, since only the operability of some trajectory vs all trajectories must be checked for each initial state. However, the problems have the same complexity (E P2hardness for Rintanen's problem is obtained by slightly adapting the proof of Theorem mentioned before), and are thus polynomially intertranslatable. Following Rintanen's and Giunchiglia's approach, finding secure plans for planning problems in K can be mapped to solving QBFs. However, since our framework is based on answer set semantics, the respective QBFs will be more involved due to intrinsic minimality conditions of the answer set semantics. Baral et al. [2000] studied the complexity of planning under incomplete information about initial states in the language A [Gelfond and Lifschitz 1993], which is similar to the framework in Rintanen [1999a] and gives rise to proper, deterministic planning domains. They show that deciding the existence of an, in our terminology, polynomial-length secure sequential plan is E P2 complete. Notice that we have considered this problem for plans of fixed length, for which this problem is DP -complete and thus simpler. From our results on the complexity of planning in the language K, similar complexity results may be derived for other declarative planning languages, such as STRIPS-like formalisms as in Rintanen [1999a] and the language A [Gelfond and Lifschitz 1993], or the fragment of C restricted to causation of literals (cf. Giunchiglia [2000]), by adaptations of our complexity proofs. The intuitive reason is that in all these formalisms, state transitions are similar in spirit and have similar complexity characteristics. In particular, our results on Secure Planning should be easily transferred to these formalisms by adapting our proofs for the appropriate problem setting.

5 Conclusion

In the report I've dealt with an approach to knowledge state planning based on nonmonototonic logic programming. The syntax and symantics of K are also introduced as per defined by the authors. The usage of K in various problems comprising of incomplete initial states is also discussed in detail here. In particular I've mentioned how knowledge states rather than world states can be represented in planning problems. For the completion the planning complexity is also discussed here.
I've given an example to enlighten the approach of problem solving using K.

6 APPENDIX:A AN EXAMPLE OF PROBLEM SOLVING

This appendix contains encoding of a well-known planning problem, which should further illustrate the practical use of language K.

6.1 The Yale shooting problem

Another example for dealing with incomplete knowledge is a variation of the famous Yale Shooting Problem (see Hanks and McDermott [1987]).We assume here that the agent has a gun and does not know whether it is initially loaded. This can be modeled as follows:
fluents : alive. loaded.
actions : load. shoot.
always : executable shoot if loaded.
executable load if not loaded.
caused - alive after shoot.
caused - loaded after shoot.
caused loaded after load.
initially : total loaded.
alive.
goal : -alive ? (1)
The total statement leads to two possible legal initial states: s1 = {loaded, alive} and s2 = {-loaded, alive}. With s1 shoot is executable, while it is not with s2. Executing shoot establishes the goal, so the planning problem has the optimistic plan =shoot= which is not secure because of s2.

7 APPENDIX:B TERMINOLOGIES USED...

Here I give a short description of the terms or principles used which may not be familiar to the readers.

7.1 The Causal Calculator

The Causal Calculator (CCalc) is a system for representing commonsense knowledge about action and change. It implements a fragment of the causal logic.

7.2 Classical Logic

Typically, a logic consists of a formal or informal language together with a deductive system and/or a model-theoretic semantics. The language is, or corresponds to, a part of a natural language like English or Greek. The deductive system is to capture, codify, or simply record which inferences are correct for the given language, and the semantics is to capture, codify, or record the meanings, or truth-conditions, or possible truth conditions, for at least part of the language.

7.3 DLV System

DLV is a deductive database system, based on disjunctive logic programming, which offers front-ends to several advanced KR formalisms

7.4 Fluents

A fluent is a function whose domain is the space Sit of situations. If the range of the function is (true, false), then it is called a propositional fluent. If its range is Sit, then it is called a situational fluent. Fluents are often the values of functions. Thus raining(x) is a fluent such that raining(x)(s) is true if and only if it is raining at the place x in the situation s. We can also write this assertion as raining(x,s) making use of the well-known equivalence between a function of two variables and a function of the first variable whose value is a function of the second variable.

7.5 Situation Calculus

Situation calculus is an aspect of the logic approach to AI. A situation is a snapshot of the world at some instant. Situations are rich objects in that it is not possible to completely describe a situation, only to say some things about it. From facts about situations and general laws about the effects of actions and other events, it is possible to infer something about future situations.

7.6 Frame Problem

The frame problem is described a stubborn difficulty arising in a first-order logic formulation, the situation calculus, in specifying which things remain unchanged when reasoning about changes in a domain. Since then, the frame problem has achieved a famous or rather notorious reputation in the Artificial Intelligence community as an example of a seemingly simple, specific problem in AI uncovering deeper and even philosophical difficulties for the task of creating artificial intelligence.

7.7 Golog Programming Language

It is a new logic programming language called GOLOG whose interpreter automatically maintains an explicit representation of the dynamic world being modeled, on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the world. This allows programs to reason about the state of the world and consider the effects of various possible courses of action before committing to a particular behavior.

References

[1] Thomas Eiter and Wolfang Faber, ACM transaction on computaional logic vol-5 No [illegible] April 2004, Pages 206-263.

[2] http reference, http://www.kr.tuwin.ac.at/staff/axel/planning/

[3] http reference, http://www.formal.stanford.edu/jme/

[4] http reference, http://citeseer.ist.psu.edu/update/548838/

[5] http reference, http://citeseer.ist.psu.edu/eiter00planning.html