Side effect analysis for logic-based planning

Kave Eshghi, Miranda Mowbray
Hewlett-Packard Laboratories,
Filton Avenue, Stoke Gifford, Bristol BS12 6QZ, England
Phone: +44 117 9228178 Fax: +44 117 9228920
{ke,mjfm}@hplb.hpl.hp.com

Abstract

In this paper we describe the algorithms for planning and, in particular, for side effect analysis used in a software tool for system management, called Dolphin, which is based on declarative programming. In Dolphin the user is given the option of declaring some side effects to be acceptable or unacceptable. This treatment of side effects makes our system different from those described in the logic-based planning literature.

Keywords: Deductive databases, Artificial Intelligence, Logic-based planning, Side effect detection and analysis

1 Introduction

In this paper we describe the planning and side effect analysis algorithms used in a software tool, called Dolphin, which is used for system management. Dolphin is a tool for helping system administrators manage complex, networked computer systems by providing them with a high level, intuitive view of the system in the context of which they can access and manipulate the the system. In Dolphin, a set of Horn clauses and integrity constraints are used to model the managed system.

These clauses and integrity constraints can be considered as the intensional part of a deductive database, while the extensional part of the database is the managed system itself (more about this later). Dolphin models are primarily used for two purposes: to query the state of the managed system, and to change its state. Seen in this light, querying the state of the managed system corresponds to querying the database, and changing the state of the system corresponds to updating the database.

In the querying mode, a top level goal is posed to the Dolphin inference engine, which is reduced to a number of extensional goals using a standard depth-first resolution strategy. The extensional goals are translated into commands which are sent to the underlying system, with the results of these commands translated back into facts which are then used to solve the extensional goals, thus completing the inference process. It is in this sense that the extensional part of the deductive database is the managed system itself.

Changing the state of the managed system is analogous to the intensional update problem for deductive databases [1], where a high level, intensional goal is posed which is translated into a number of lower level, extensional updates. In fact, the first part of the algorithm we use for this purpose is similar to the abductive algorithms developed for intensional updates. There is another dimension to our problem, however. In deductive databases, the updates are independent, and don't have preconditions or side effects. In our case, we don't have updates. We have actions which, when performed on the underlying system, will bring about the desired update. But actions have preconditions and side effects associated with them, which makes it necessary to take into account the interaction between the actions and their side-effects, and the consequences of the side effects on the rest of the system. This paper discusses some of these issues, with emphasis on the side-effect detection and amelioration aspects.

2 Side Effects

Suppose you want to move from the living room to the kitchen next door. The door is closed. You could open the door and walk through; or you could take a battering ram and break through the wall. Both sequences of actions will get you to the kitchen, but one will have more serious side effects than the other. Some consequences of your actions will be unavoidable if you are to attain your goal - for example, if you are in the kitchen, you will no longer be in the living room. Some consequences will be not inevitable, but tolerable - for instance, you might not mind that the door handle squeaks when you turn it. Some consequences can be eliminated - for example, you may not like the door being open, but if so, you can shut it again once you're in the kitchen. Finally, some consequences will be undesirable and difficult to eliminate - such as the destruction of the wall.

In the classic literature on logic-based planning, (eg. [3], [4]) there are two attitudes to side effects. The first is that a plan should produce the goal with NO side effects, in which case there is no plan which will take you from the living room to the kitchen. The second is that as long as the goal is reached, side effects do not matter, in which case you will have to extend your original simple goal of moving from one room to another, or else you may end up minus a wall. We argue that for some applications of computer-assisted planning it is more natural and flexible to allow some side effects, but not others, and to determine via a dialogue with the user which of the side effects are tolerable. In this paper we describe a way to achieve this.

3 Theoretical Framework

In this section we present the theoretical framework of the planning/side effect analysis problem. The logical language used is first order predicate calculus in Clausal Normal Form, of which Horn Clauses are a subset.

1. There is a theory T, which is a set of Horn clauses, and a set of integrity constraints I, where each integrity constraint is a clause. (Although we do not have negation as failure in the language, it can be simulated by an abductive style transformation. We omit the details here, since this is not the point of the paper).

   An integrity constraint is allowed to be broken temporarily in the middle of a plan, but must be satisfied by the state reached at the end of a plan.

2. There is a set B of *extensional predicates*; this is just the set of predicates in the language of the Horn clause theory which do not occur in the head of any clause in T. The set of positive ground atoms whose predicate is in B is denoted by B^+.

3. There is a set A of possible actions, each action $a \in A$ has a set $Pre(a)$ of preconditions and a set $Post(a)$ of postconditions. All the conditions in $Post(a)$ are in $B^+ \cup \{\neg b^+ : b^+ \in B^+\}$.

4. There is an initial state s_0, which is a subset of B^+ such that s_0+T+I is consistent. The interpretation of s_0 is that it is the state of the system in which a condition is true iff it is true in the minimal model of T+s_0.

5. There is a goal G.

A sequence of actions $< a_1, a_2, \ldots, a_k >$ *induces*
a sequence of states $< s_0, s_1, \ldots, s_k >$ iff for each $1 \leq i \leq k$,
$s_i = (s_{i-1} \cup$ (positive atoms in $Post(a_i)) \setminus$ (positive atoms whose negation is in $Post(a_i)))$
A sequence of actions $< a_1, a_2, \ldots, a_k >$ is *possible* iff it induces $< s_0, s_1, \ldots, s_k >$ and for each i, each condition in $Pre(a)$ is true in the minimal model of T+s_{i-1}.
A sequence of actions $< a_1, a_2, \ldots, a_k >$ *satisfies G* iff it induces $< s_0, s_1, \ldots, s_k >$, it is possible, and G and I are true in the minimal model of T+s_k. In general there may be more than one plan which satisfies G. (Or there may be none.)

4 Example

This section gives an example of the kind of planning problem that we will address in this paper. It is a simplified version of a planning problem that the authors encountered when using the techniques described in this paper to do remote administration of a UNIX workgroup. Variables are written in upper case.

There are two main kinds of objects considered in this example, users and files. The files are uniquely identified by their pathnames, which will be denoted by the variable PN. Users are uniquely identified by their user IDs. Users also have names, denoted by the variable N, which can change. A file can be read by a user if it is owned by that user, if it is world-readable, or if the user is super-user.

Theory T is the theory
canRead(N,PN) ← ownsFile(N,PN)
canRead(N,PN) ← isUserName(N), isFile(PN), readability(PN,*world*)
canRead(N,PN) ← isFile(PN), isSuperUser(N)
ownsFile(N,PN) ← userID(N,ID), IDofOwner(PN,ID)
isUserName(N) ← userID(N,ID)
isFile(PN) ← IDofOwner(PN,ID)

A file can be owner-readable or world-readable. It is possible to put a status lock on a file, for safety reasons; a file can change between the states owner-readable and world-readable only if its status is unlocked. For security reasons, there is an integrity constraint expressing the condition that at the end of a sequence of actions the only user who can be super-user is the user with user ID 0.

Integrity Constraints The set of integrity constraints I is
readability(PN,*owner*) ∨ readability(PN,*world*) ← isFile(PN)
(ID1=ID2) ← readability(PN,ID1), readability(PN,ID2)
(ID1=ID2) ← userID(N,ID1), userID(N,ID2)
(N1=N2) ← userID(N1,ID), userID(N2,ID)
(ID1=ID2) ← IDofOwner(PN,ID1), IDofOwner(PN,ID2)
userID(N,0) ← isSuperUser(N)

Extensional predicates The set of extensional predicates B is
{readability(PN,ID), userID(N,ID), IDofOwner(PN,ID),
isSuperUser(N), statusLocked(PN)}.

Actions

The actions that are available include the following.
(Strictly speaking, the descriptions below are not of actions but of action schemata; they are turned into actions by substituting constants for each variable.)

statusLock(PN)
Pre: ¬statusLocked(PN). Post: statusLocked(PN)

statusUnlock(PN)
Pre: statusLocked(PN). Post: ¬statusLocked(PN)

makeWorldReadable(PN,READ)
Pre: readability(PN,READ), (READ ≠*world*), ¬statusLocked(PN).
Post: ¬readability(PN,READ), readability(PN,*world*)

makeSuperUser(N)
Pre: ¬isSuperUser(N), isUserName(N). Post: isSuperUser(N)

changeFileOwner(PN,ID1,N,ID2)
Pre: IDofOwner(PN,ID1), userID(N,ID2), (ID1 ≠ ID2).
Post: IDofOwner(PN,ID2), ¬IDofOwner(PN,ID1)

In the initial state, there is a file with pathname *pn*; the goal is to make this file readable by the user with name *miranda*.

Initial state The initial state s_0 includes the conditions
IDofOwner(*pn*,15), userID(*kave*,15), userID(*miranda*,10),
statusLocked(*pn*), readability(*pn*,*owner*).
Goal The goal G is just the condition canRead(*miranda*,*pn*).

5 Generating a Sequence of Actions

Generating a sequence of actions to satisfy the goal is a two step process: first, through back chaining, the top level goal is reduced to a number of extensional goals which, if satisfied, would imply the top level goal. Then a planner is used to find a sequence of actions at the end of which the set of extensional goals will be satisfied.

Our action planning system is different from planning systems described in the literature [2] [3] [4] due to the following requirements, which add complexity to the planning process:
1) Actions only have extensional postconditions, but the goal and the preconditions of actions can be in terms of intensional predicates.
2) We allow integrity constraints which are in general expressed in terms of intensional predicates. The sequence of actions generated must be such that at the end of it, none of the integrity constraints are violated.

In the example, we start from the goal ← canRead(*miranda*,*pn*)
and we resolve it with the clause canRead(N,PN) ← ownsFile(N,PN)
we are left with the goal ← ownsFile(*miranda*,*pn*)
We then resolve this goal with the clause
ownsFile(N,PN) ← userID(X,ID), IDofOwner(PN,ID)
which gives us the goals ← userID(*miranda*,ID), IDofOwner(*pn*,ID)
which, when resolved with the assertion userID(*miranda*,10)
in the initial state, will give the goal ← IDofOwner(*pn*,10)
as the residue. Then we invoke the planner to generate a sequence of actions to satisfy this residual goal. The planner would generate the action sequence
<changeFileOwner(*pn*,15,*miranda*,10)> to satisfy this goal.

Notice that there are two levels of non-determinism in the planning process as described above. Firstly, there is non-determinism in the reduction of the top level goal to extensional goals. For example, if we had chosen the clause
canRead(N,PN) ← isUserName(N), isFile(PN), readability(PN,*world*)
to resolve with the top level goal, we would have ended up with a different set of extensional goals to be satisfied by the action generator. Secondly, there is the traditional non-determinism associated with choosing actions to satisfy the extensional goals. Although in this example the possible actions are unique, in general there can be more than one possible action or sequence of actions to satisfy the given set

of extensional goals.

It is not our purpose to give the details of the planning algorithm in this paper, and the description above is included to provide a context for the side-effect detection algorithm.

6 Detecting side effects

In our notation we list the postconditions of each action. These postconditions will be the consequences that we consider. We don't attempt to describe *all* the consequences of an action, just ones whose effects are necessary for planning purposes and/or may be considered undesirable by the user. For example, changing the owner of a file will involve changing some data base entry, and may increase or decrease the number of bytes of data in the data base, but this effect is not recorded as a postcondition of the action changeFileOwner(PN,ID1,N,ID2).

We consider the *reportable side effects* of a sequence of actions to be those logical statements which are in the union of the sets of postconditions of the actions, which are true in the final state, which were not true in the initial state, and which are not direct consequences of the goal (ie. they are not logical consequences of T+G+I). When we consult the user, it is these statements that we will present.

The choice of this set of statements rather than another to be the side effects that we report to the user is to some extent a matter of taste. It could be argued, for instance, that if a statement is originally true, becomes false during the course of the sequence of actions, and then is made true again, then it should be reported as a side effect; we do not do this, because we do not report anything which was true in the original state. Moreover, it is possible that some of the statements we report will be true in any plan which achieves the goal, although they are not logical consequences of T+G+I. We choose to report such statements, because they may be undesirable to the user. It is safer to give the user the chance of rejecting all plans to move into the kitchen if there really is no way to do it from the given initial state without knocking down the wall, than to go ahead with the battering ram.

How to calculate the reportable side effects

Given a sequence of actions $< a_1, \ldots, a_k >$ satisfying G, it is straightforward to calculate the reportable side effects. This section gives a not particulary efficient, but simple, algorithm which does this calculation.

The algorithm takes as input not only the sequence of actions and G, but also a set *Allowed side effects*. If there has been no communication yet with the user, this set is empty. Conditions are added to it by the algorithm. After each iteration of the algorithm the user has the option of designating some of the reported side effects as OK and not worth reporting, and others as unacceptable. (The remaining side effects may be generated by future plans, but if they are they will be reported to the user.) The side effects that the user indicates are OK and not worth reporting are added to the set *Allowed side effects*. A new goal is derived, which is just the old goal plus the negations of all the unacceptable side effects. The new goal is fed into the plan generator, which comes up with a sequence of actions to satisfy the new goal (if it can find one); this new sequence of actions is used as input for another iteration of the algorithm to find the reportable side effects, these effects are reported to the user, and so on until the user is satisfied or no sequence satisfying the goal is found.

1. Initialize the set R to the empty set, and *counter* to k.

2. Set $P = Post(a_{counter}) \setminus R$.

3. For each $post \in P \setminus$ *Allowed side effects* such T+G+I proves *post*, add *post* to *Allowed side effects* and to R, and remove it from P.

4. Pick $post \in P$. If neither *post* nor $\neg post$ are in R, then add *post* to R.

5. Remove *post* from P. If P is nonempty, return to step 4. Otherwise go on to step 6.

6. If *counter*> 1 then decrease *counter* by one and return to step 2; else go on to step 7.

7. For each *post* in $R \cap$ *Allowed side effects*, remove *post* from R.

8. For each *post* in $R \cap s_0$, remove *post* from R. For each negative $post \in R$ such that $\neg post \notin s_0$, remove *post* from R.

9. R is now the set of the reportable side effects. Report it to the user.

It is straightforward to check that at the beginning of step 7 the set R contains exactly the conditions *post* which are not direct effects of the goal and are a post-condition of some action a_j where $\neg post$ is not a postcondition of any of the actions $a_{j+1}, a_{j+2}, \ldots, a_k$. It follows from the definition of s_k that the side effects reported to the user are exactly those conditions that are in a postcondition of one of the actions, that are true in the state represented by s_k and false in the state represented by s_0, and that are not direct effects of the goal G.

The user has the option of adding conditions to the set *Allowed side effects*, or changing the goal. There are occasions when changing the goal may be particularly useful. For example, suppose that a plan is generated with reported side effect IDofOwner(pn,1). The user says that this is unacceptable, because he or she doesn't want the ownership of the file with pathname pn to move from its original owner, who has user ID 15. A new plan is generated with reported side effect IDofOwner(pn,2). The user doesn't like this either and a new plan is generated with reported side effect IDofOwner(pn,3). The user now spots a pattern and adds the condition IDofOwner(pn,15) to the goal. The effect of adding conditions to the set *Allowed side effects* is that these will not be reported if they arise as side effects. This doesn't change the plans that are generated, but can make life simpler for the user by ensuring that irrelevant information is suppressed.

An optional way of further simplifying the data reported to the user is to remove side effects which are redundant because they are implied by other reported side effects together with the integrity constraints. To do this, pick *post* in R and check whether G+$R \setminus \{post\}$+$\neg post$ violates any of the integrity constraints I. If it does,

remove *post* from R. Pick a new *post* in R which has not yet been checked, and repeat, until all members of R have been checked.
This can be computationally complex, and suppresses the reporting of some side effects which the user may actually want to know about, so we do not make it an integral part of the side detection algorithm.

Using the planner and side effect analyser together

One advantage of the algorithm given above is that is possible to use it incrementally, in conjunction with the plan generator. Suppose the plan generator produces a sequence of actions which satisfies the goal starting from an initial state which is not s_0 but some other state s'_0. Steps 1-5 of the algorithm for side effect detection can be used to find an interim set R' of side effects, equal to the value of R after these steps. R' is stored for later reference. The plan generator can then produce two different sequences of actions which satisfy s'_0 starting at initial state s_0. Now the side effects of the two different plans to satisfy G (which have the same ending but different beginnings) can be calculated, by setting R equal to R' and applying steps 2-9 of the algorithm to the two sequences of actions to satisfy s'_0. More complicated backtracking manoevres during the planning can similarly be followed without too much recalculation by the side effect detector, by appropriate storage of partial results.

7 Discussion of the example

The goal of the example can be made true, for example, by
– transferring ownership of the file from *kave* to *miranda*;
– turning off the status lock and then changing the file to world-readable;
– making *miranda* super-user - but this violates one of the integrity constraints.

Suppose that the user had in mind changing the readability, rather than changing the file's owner. Here is an example of the steps that could be gone through by the user, planner and side effect analyser;

1. Planner generates the simple plan <changeFileOwner(*pn*,15,*miranda*,10)>
2. Side effect analyser reports side effects IDofOwner(*pn*,10), $\neg$IDofOwner(*pn*,15) to the user
3. User says that the effect $\neg$IDofOwner(*pn*,15) is unwanted. The new goal canRead(*miranda*,*pn*), IDofOwner(*pn*,15) is sent back to the planner.
4. Planner generates <statusUnlock(*pn*), makeWorldReadable(*pn*,*owner*)>.
5. Side effect analyser reports side effects
$\neg$statusLocked(*pn*), $\neg$readability(*pn*,*owner*), readability(*pn*,*world*).
(If the optional step removing redundant effects from the set of reported effects were used, then $\neg$readability(*pn*,*owner*) would not be reported.)
6. User says that the effect $\neg$statusLocked(*pn*) is unwanted, but that the other two are OK. The conditions $\neg$readability(*pn*,*owner*), readability(*pn*,*world*) are added to

the set *Acceptable side effects*, and the goal canRead(*miranda*,*pn*),
IDofOwner(*pn*,15), statusLocked(*pn*)
is sent back to the planner.
7. Planner generates
<statusUnlock(*pn*), makeWorldReadable(*pn*), statusLock(*pn*)>
8. Side effect analyser calculates that there are no reportable side effects. So the plan is fine. A note of the plan is sent to the user; the plan is scheduled.

This procedure for generating a sequence of actions which will satisfy the goal may not in theory terminate. In practice, there is a parameter (which the user can change, default is 10) which is used to bound the number of actions in a sequence. If the procedure fails to find any sequences with length less than the parameter satisfying the goal, the search is terminated. Sequences with too many actions are undesirable because uncertaincies about the exact effects of actions are cumulative when actions are performed in sequence.

References

[1] Bry, F. *Intensional updates: abduction via deduction* Proc. ICLP 90

[2] Denecker, M., Missiaen, L., Bruynooghe, M., *Temporal reasoning with Abductive Event Calculus* Proc. ECAI 92

[3] Chapman, D., *Planning for conjunctive goals* Artificial Intelligence Vol. 32, 1987

[4] Fikes, R. E. & Nilsson, N. J *STRIPS: a new approach to the application of theorem proving to problem solving*, Artificial Intelligence Vol. 2, 1971

[5] Gelfond, M. & Lifschitz, V. *Representing actions in extended logic programming* Proc. JICSLP 92

[6] Kowalski, R. A. *Database updates in Event Calculus* The Journal of Logic Programming 12

[7] Phan Minh Dung, *Representing actions in logic programming and its application in database updates* Proc. ICLP 93