Abductive Partial Order Planning with Dependent Fluents

Liviu Badea, Doina Tilivea

AI Lab, National Institute for Research and Development in Informatics
8-10 Averescu Blvd., Bucharest, Romania
badea@ici.ro

Abstract. Our query planning application for system integration requires a backward partial-order planner able to deal with non-ground plans in the presence of state constraints. While many partial-order planners exist for the case of independent fluents, partial-order planning with *dependent fluents* is a significantly more complex problem, which we tackle in an abductive event calculus framework. We show that existing abductive procedures have *non-minimality* problems that are significant especially in our planning domain and propose an improved abductive procedure to alleviate these problems. We also describe a general transformation from an abductive framework to *Constraint Handling Rules (CHRs)*, which can be used to obtain an efficient implementation.

1 Introduction and motivation

The integration of hybrid modules, components and software systems, possibly developed by different software providers, is a notoriously difficult task, involving various extremely complex technical issues (such as distribution, different programming languages, environments and even operating systems) as well as conceptual problems (such as different data models, semantic mismatches, etc.).

Solving the conceptual problems requires the development of a common, explicit, declarative knowledge model of the systems to be integrated. Such a model should be used (by a so-called *mediator*) not only during the development of the integrated system, but also during runtime, when it can be manipulated by an intelligent *query planning* agent to solve problems that could not have been solved by any of the specific information sources alone and might even not have been foreseeable by the system integrator. Since in most realistic applications, the state of the databases or of the procedural components changes as a problem is being solved, we shall describe the services offered by such procedural applications and the database updates as *actions*.

The *query planner* of the mediator transforms a user query into a partially ordered sequence of information-source specific queries, updates and calls to application interfaces that solve the query. The main requirements which our system integration application imposes on the query planner are the following:

- It should be a *partial-order planner* in order to take advantage of the intrinsic distributed nature of the integrated system.

- As the information content of the sources (for example databases) can be quite large, forward propagation of the initial state of the sources is impossible in practice. We therefore need to develop a *backward* planner, which would ensure minimal source accesses with maximally specific queries.
- The planner should reason with *arbitrary logical constraints* between state predicates (*dependent* fluents).
- It should also be able to manipulate *plans with variables*. UCPOP-like planners can do this, but UCPOP only deals with *independent* fluents.

Also, although very fast general purpose planners like *Graphplan* and *SATPLAN* are currently available, these are not usable in our system integration application, where generating a grounding of the planning problem is inconceivable (a database may contain an enormous number of different constants), while both *SATPLAN* and *Graphplan* generate a grounding of the planning problem. The same holds for recent planners based on Answer Set Programming developed in the Logic Programming community, which use efficient *propositional* answer set generators, like *dlv* or *smodels*.

Unfortunately, there is no implemented planner available with the characteristics required by our system integration application. In this paper we describe the construction of such a planner.

The following simple example illustrates the type of problems we are dealing with. Assume that the dean of a university plans to assign a high responsibility course (this course being assignable only to faculty members). This can be done by applying action assign_course having effect course_assigned and no explicit preconditions. Assume there exist constraints, such as that this course cannot be assigned to a person who is not a professor

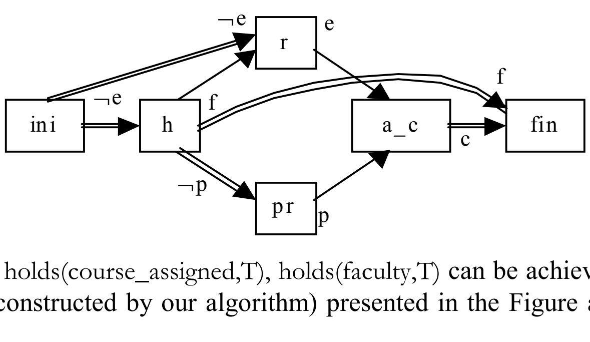
$$\leftarrow \text{holds(course\_assigned, T), holds}(\neg\text{professor, T}) \quad (1)$$

or who is not employed by the university

$$\leftarrow \text{holds(course\_assigned, T), holds}(\neg\text{employed, T}) \quad (2)$$

Three additional actions can be used to achieve the final goal:

- hire (with precondition ¬employed and effects faculty and ¬professor) hires a person that is not registered in the personnel database as employed (this person may not be directly hired as a professor),
- promote (with precondition ¬professor and effect professor) which promotes a non-professor to the position of professor,
- register (with precondition ¬employed and effect employed) which registers a person as being employed in the database of the personnel department.



The final goal holds(course_assigned,T), holds(faculty,T) can be achieved by the partial-order plan (constructed by our algorithm) presented in the Figure above. Double arrows denote protection intervals not_clipped(T_i, F, T_f), while simple arrows are plain ordering constraints. Note that state constraints induce implicit preconditions as well as ordering constraints and protection intervals. For example, the ordering constraints r < a_c and pr < a_c have been posted to prevent the activation of the state constraints (2) and (1) respectively.

2 Problems with abductive procedures

Partial-order planning can be viewed as abductive planning in the Event Calculus [7]. Considering the fact that there are many partial-order planners for *independent* fluents, and since adding state constraints to an Event Calculus action specification seems straight-forward (see (8) below), it might seem that it should be easy to develop a partial-order planner dealing with *dependent* fluents.[1] This impression is however misleading: integrity constraints represent a significant complication. Intuitively, fluent dependencies seriously complicate the detection of '*threats*'. As far as we know, there are no implemented sound and complete partial-order planners able to deal with *non-ground plans* (i.e. plans with existentially quantified variables) and *dependent fluents* exist.[2] (Dependent fluents are fluents subject to integrity constraints. As usual, integrity constraints may have universally quantified variables.)

In the following we illustrate problems faced by existing abductive procedures, especially in our planning domain.

We start with the following (simplified[3]) *Abductive Logic Programming (ALP)* specification of the Event Calculus with dependent fluents:

$$
\begin{array}{ll}
P\left\{\begin{array}{l}
\text{holds}(F,T) \leftarrow \text{starts}(F,T_0),\ T_0<T,\ \text{not clipped}(T_0,F,T) \\
\text{clipped}(T_0,F,T) \leftarrow \text{starts}(\neg F,T_1),\ T_0<T_1,\ T_1<T \\
\text{starts}(F,0) \leftarrow \text{initially}(F) \\
\text{starts}(F,T) \leftarrow \text{initiates}(A,F),\ \text{happens}(A,T)
\end{array}\right. & \begin{array}{r}(3)\\(4)\\(5)\\(6)\end{array} \\
I\left\{\begin{array}{l}
\leftarrow \text{happens}(A,T),\ \text{precondition}(A,F),\ \text{not holds}(F,T) \\
\leftarrow \text{holds}(F_1,T),\ \ldots,\ \text{holds}(F_n,T)
\end{array}\right. & \begin{array}{r}(7)\\(8)\end{array}
\end{array}
$$

Abducibles A={happens}
Query: ?- holds(F_i,T_i), …, holds(F_j,T_j), … $T_i<T_j$, …

where P are program clauses, while I are integrity constraints. A denotes actions, T time points and F fluent literals, i.e. (negated) atoms. starts(F,T) means that fluent F becomes true at T, while starts(¬F,T) means that F is terminated at T. Direct effects are given by initiates, while preconditions are given by precondition.

[1] Some implementations (including UCPOP) distinguish between *primitive* and *derived* fluents. (Primitive fluents are assumed *independent* and actions cannot have derived fluents as effects.) This is a simple form of *ramification* – the primitive fluents are still independent (state constraints are not allowed).

[2] For example, the planner of [7] is unsound in the presence of state constraints (see the last example in section 6 of [7]). [1] has non-minimality problems (see below) and we haven't been able to use it for dependent fluents.

[3] More complicated action languages can be easily incorporated.

(8) represents a problem-dependent integrity constraint involving fluents F_1, …, F_n.

A purely backward abductive procedure (like *ACLP* [4] or *SLDNFA* [2]) typically faces problems in recognizing that an abducible selected for solving one subgoal also solves a second subgoal. If the second subgoal can be achieved in several ways, the first solution returned by the backward abductive procedure might be non-minimal, since it might *reachieve an already achieved goal*. For example, consider the following ALP $P = \{p \leftarrow a,\ q \leftarrow b,\ q \leftarrow a\}$ with a, b abducibles and the query ?- p, q. For solving the first subgoal, p, the abductive procedure will abduce a. Then, when trying to solve q, it will abduce b, not realizing that q has already been solved by assuming a. (Both *ACLP* and *SLDNFA* have this problem, the latter even if a, b are declared strong abducibles.)

This problem could be solved by forward propagation of a to q: $a \Rightarrow q$. Now, when trying to solve q, the procedure would find that q has already been achieved.

Such situations occur very frequently in our planning domain, where an effect can be achieved by several actions. For example, if {initiates(a_2,f_2), initiates(a_1,f_1), initiates(a_1,f_2)}, then the first answer to the query ?- holds(f_1,t), holds(f_2,t) will be non-minimal (both a_1 and a_2 will be applied).

There is a second situation in which existing abductive procedures might produce non-minimal solutions, namely the presence of *negative literals in negative goals*. As these are in fact disjunctions in disguise, existing abductive procedures will treat them by *splitting*, even in cases in which the negative goal is already achieved. For example, for the ALP $P = \{p \leftarrow a,\ q \leftarrow b,\ r \leftarrow b\}$ with abducibles $A = \{a, b\}$ and integrity constraints $I = \{ \leftarrow \text{not } p, \text{not } q \}$, the query ?- r would first lead to abducing b for solving r, and then generate the negative goal $\leftarrow$ not p, not q. Because b entails q, this goal is already solved, but a backward abductive procedure would not know this and would split the negative goal into the positive goal p and the negative goal $\leftarrow$ not q (which will be reduced to q). Unfortunately, this would abduce a to achieve p, leading to a *non-minimal* solution.[4] (In other words, we are "repairing" an already repaired integrity constraint.) Again, propagating b forward to q would allow us to inactivate the negative goal $\leftarrow$ not p, not q before splitting it.

This situation does not arise in planning with independent fluents (i.e. without state constraints). Several negated literals can appear in negative goals only due to the state constraints (8) (after unfolding holds to not clipped). The presence of state constraints therefore complicates partial-order planning to a significant extent. (The solution to the frame problem offered by partial-order planning is especially simple in the case of *independent* fluents.)

For example, consider the actions {initiates(a_1,¬p), initiates(a_2,¬q), initiates(a_2,r), initiates(b_1,p), initiates(b_2,q), initiates(b_3,s)}, the state constraint

$$\leftarrow \text{holds}(p,T), \text{holds}(q,T), \text{holds}(s,T)$$

and the goal ?- holds(r,t_{fin}),…. Further assume that actions a_2, b_1, b_2, b_3 solve all positive goals and have already been placed in the plan: happens(a_2,t), happens(b_1,t_1), happens(b_2,t_2), happens(b_3,t_3), together with the ordering constraints $t_1 < t_3$,

4 *SLDNFA* has this problem too. Due to its syntactical restrictions, *ACLP* cannot even deal directly with this example.

$t_2<t_3$, $t_2<t$, $t<t_3$. Now, the integrity constraint will be successively unfolded to the negative goals:

$\leftarrow$ happens(b_1,T_1), not clipped(T_1,p,T), $T_1<T$,
  happens(b_2,T_2), not clipped(T_2,q,T), $T_2<T$,
  happens(b_3,T_3), not clipped(T_3,s,T), $T_3<T$.
$\leftarrow$ not clipped(t_1,p,T), not clipped(t_2,q,T), not clipped(t_3,s,T),
  T>max(t_1,t_2,t_3).

(the last step corresponds to the resolution with the abducibles happens(b_i,t_j)).
The last negative goal is already solved because the presence of a_2 in the plan ensures that clipped(t_2,q,t_3) holds. Solving the negative goal by splitting, for example by generating the positive subgoal ?- clipped(t_1,p,T), T>max(t_1,t_2,t_3) would place the additional unnecessary action a_1 in the plan.

A careful analysis of existing abductive procedures shows that both problems mentioned above involve reachieving an already achieved goal and are related to the treatment of implicit disjunctions by splitting.

We argue that in both cases we should avoid splitting disjunctions when these are already achieved. This will not *guarantee* the minimality of the *first* solution, but it will at least avoid reachieving already achieved goals. Of course, the minimal solution is in the search space, but in general we cannot guarantee obtaining it in polynomial time. More precisely, the problem of finding a (locally) minimal[5] explanation in an abductive problem with integrity constraints is *NP*-complete, even in the propositional case (Theorem 4.5 of [10]).

For example, considering the same ALP program P = {p $\leftarrow$ a, q $\leftarrow$ b, q $\leftarrow$ a} as above, the query ?- q, p will lead in our framework to the non-minimal abductive explanation b,a. Note however, that the minimal solution a is in the search space and will be found upon backtracking.

3 Propagating abductive changes

We have seen that the planning problem can be formulated as an abductive problem. The main efficiency problem faced by all implemented abductive procedures is avoiding testing *all* integrity constraints after each abductive change. Since the integrity constraints have been tested before the change, we should retest only the ones that are influenced by the change in some abducible. For example, for achieving this, *ACLP* [4] requires each integrity constraint (IC) to contain at least an abducible predicate. The current implementation also requires each non-abducible positive condition in an IC not to depend on abducible predicates. If it does, as it is usually the case, the user would have to unfold the predicate in the IC with its definition until its dependence on the abducibles is made explicit. These strong requirements are needed so that there are only *direct* influences of changing abducibles on ICs. If this limitation is removed, then we need to be able to determine which predicates are (indirectly) influenced by a change in an abducible. This can be achieved by *forward propagation* of the abductive changes from abducibles to other predicates occurring in ICs.

[5] *parsimonious* in the terminology of [10].

In the following we propose a mixed abductive procedure combining *backward* goal reduction rules with *forward* propagation rules for the abductive changes. In this problem solving strategy, the goals are reduced backwards to abducibles and constraints, which are then propagated forward ("saturated" to a complete solution). The role of the forward propagation rules is not only to detect inconsistencies, but also to *repair* any *potential* inconsistencies (by adding new abducibles and/or constraints). Instead of retesting all ICs after each modification, we propagate the change through the ICs and suggest repairs that ensure that the ICs are not violated.

3.1 Constraint Handling Rules

Constraint Handling Rules (CHRs) [3] represent a flexible approach to developing user-defined constraint solvers in a declarative language. As opposed to typical constraint solvers, which are black boxes, CHRs represent a 'no-box' approach to CLP. CHR propagation rules are ideal candidates for implementing the rules for forward propagation of abductive changes.

CHRs can be either *simplification* or *propagation* rules.

A *simplification* rule Head ⇔ Guard | Body replaces the head constraints by the body provided the guard is true (the Head can contain multiple CHR constraint atoms).

Propagation rules Head ⇒ Guard | Body add the body constraints to the constraint store without deleting the head constraints (provided the guard is true). A third, hybrid type of rules, *simpagation rules* $\mathrm{Head}_1 \setminus \mathrm{Head}_2$ ⇔ Guard | Body replace Head_2 by Body (while preserving Head_1) if Guard is true. (Guards are optional in all types of rules.)

4 Transforming ALPs into CHRs

In the following, we present a general transformation from an Abductive Logic Program (ALP) into a set of Constraint Handling Rules (CHRs), which function as an abductive procedure for the given ALP. We also illustrate the transformation on the example of partial-order planning with dependent fluents.

We start from an ALP $\langle P,A,I \rangle$. In the case of partial-order planning, we will use the ALP (3)-(8) from Section 2. Our goal is to replace the integrity constraints (ICs) (which would naively have to be retested after each abductive step) by forward rules for propagating abductive changes. This is useful both for detecting inconsistencies and for suggesting repairs. However, not every predicate in an IC can be the target of forward propagation - such predicates would have to be unfolded (backwards) until "forward" predicates are reached.

Since the specific problem may require certain predicates or rules to be "backward" -- even if they *could* in principle be "forward" -- we allow them to be *explicitly declared as "backward"*.[6] (This is obviously a problem-dependent specification.)

The transformation rules below will automatically determine the status (forward/backward) of the predicates and rules that are not explicitly declared as "backward".

Note that the transformation of ALPs into CHRs presented below is completely general (it works for any ALP program, not just for our planning domain).

$\boxed{1}$ First, we determine which rules and predicates can be treated by forward propagation (or, in short, are *"forward"*). (Rules and predicates which are not "forward" are called *"backward"* and will have to be treated by unfolding.)

- *Abducibles* (like happens in our planning domain) are automatically "forward". The occurrences of such abducibles p in rule bodies will be replaced by constraints $\mathcal{C}$p. (Intuitively, $\mathcal{C}$p denotes the "open" part of predicate p. Technically, $\mathcal{C}$p is used to trigger the forward propagation rules for p -- see (*) below).
- An *ALP rule* can be "forward" only if all its body literals are "forward". In particular, rules with negative literals in the body cannot be used as forward rules (for ex. rule (3)).
- Predicates which appear in the head of at least one "forward" ALP rule are themselves "forward".

$\boxed{2}$ Then, we replace the ICs by forward propagation rules. For each IC, we unfold the positive literals - in all possible ways *with their "backward" rules only* - until we are left with positive "forward" literals, negative literals or constraints:

$$\leftarrow p_1, \ldots, p_n, \text{not } q_1, \ldots, \text{not } q_m, c_1, \ldots, c_k$$

Such an unfolded IC is replaced by the forward propagation rule

$$\mathcal{C}p_1, \ldots, \mathcal{C}p_n \Rightarrow \sim c_1 ; \ldots ; \sim c_k ;\ c,(q_1 ; \ldots ; q_m) \qquad (*)$$

where $\sim c_i$ is the complement of the constraint c_i and c is the conjunction $c_1, \ldots, c_k$ (we apply a sort of semantic splitting w.r.t. the constraints).

This forward rule exactly captures the functioning of the abductive procedure, which waits for $p_1, \ldots, p_n$ and only then treats the remaining body by splitting. Note that the *subgoal* q_j is propagated (rather than the *constraint* $\mathcal{C}q_j$). Solving the subgoal q_j amounts to constructively ensuring that the IC is not violated (this functions as a *constructive "repair"* of a *potential* IC violation).

$\boxed{3}$ Finally, we replace negative literals not p in bodies of "backward" rules by constraints $\mathcal{C}$not_p (whose role will be to protect against any potential inconsistencies with some p).[7] For each such negative literal we add the IC $\leftarrow$ not_p, p, which will be treated as in step (2) above.

[6] For example, in our planning domain, we may wish to avoid propagating the initial state forward, especially if we are dealing with a database having a huge number of records.

[7] Unlike "normal" abducibles which are implicitly "minimized", abducibles of the form $\mathcal{C}$not_p are subject to a maximization policy. Thus, we cannot expect *all instances* $\mathcal{C}$not_p to be ex-

In our **planning domain**, rule (3) is backward because it has a negative literal (not clipped) in the body.

Rules with constraints[8] in the body could be used as forward rules, but they would propagate disjunctions (treated by splitting), which should be avoided. (4) will therefore be treated backward because of the constraints '<' in its body.

(5) will be treated backward because propagating the initial state of a database for example (having a huge number of records) is infeasible in practice.

(6) can be treated as forward, so starts will be a "forward" predicate: [9]

$$\mathbf{C}\text{happens}(A,T),\ \text{initiates}(A,F) \Rightarrow \mathbf{C}\text{starts}(F,T) \qquad (9)$$

Similarly, the IC (7) induces the forward rule

$$\mathbf{C}\text{happens}(A,T),\ \text{precondition}(A,F) \Rightarrow \text{holds}(F,T) \qquad (10)$$

(The action description predicates precondition and initiates are "static constraints", i.e. constraints that are given at the beginning of the problem solving process and which do not change.)

Finally, the ICs (8) related to state constraints are unfolded (since holds is a backward predicate) to

$$\leftarrow \text{starts}(F_1,T_1),\ \text{not clipped}(T_1,F_1,T),\ T_1<T,\ \ldots,$$
$$\text{starts}(F_n,T_n),\ \text{not clipped}(T_n,F_n,T),\ T_n<T.$$

which induce the forward rules

$$\mathbf{C}\text{starts}(F_1,T_1),\ \ldots,\ \mathbf{C}\text{starts}(F_n,T_n) \Rightarrow T>\max(T_1,\ \ldots,T_n), \qquad (11)$$
$$(\ \text{clipped}(T_1,F_1,T)\ ;\ \ldots\ ;\ \text{clipped}(T_n,F_n,T)\ ).$$

(The first disjunct, $T<\max(T_1, \ldots,T_n)$, of the consequent could be dropped since it mentions the free variable T.)

Since starts also has a backward rule (5), the IC (8) also unfolds to

$$\leftarrow \ldots,\ \text{initially}(F_{ik}),\ \text{not clipped}(0,F_{ik},T),\ \ldots,$$
$$\ldots,\ \text{starts}(F_{jl},T_{jl}),\ \text{not clipped}(T_{jl},F_{jl},T),\ T_{jl}<T,\ \ldots$$

The induced propagation rule is

$$\ldots,\ \mathbf{C}\text{starts}(F_{jl},T_{jl}),\ \ldots \Rightarrow \ \ldots\ ;\ \sim\text{initially}(F_{ik})\ ;\ \ldots\ ; \qquad (12)$$
$$\text{initially}(F_{i1}),\ \ldots,\ \text{initially}(F_{im}),\ T>\max(T_{j1},\ \ldots),$$
$$[\ \ldots\ ;\ \text{clipped}(0,F_{ik},T)\ ;\ \ldots\ ;\ \ldots\ ;\ \text{clipped}(T_{jl},F_{jl},T)\ ;\ \ldots\ ]$$

Finally, the IC

$$\leftarrow \text{not\_clipped}(T_0,F,T),\ \text{clipped}(T_0,F_1,T),\ F=F_1$$

is unfolded with (4) to

$$\leftarrow \text{not\_clipped}(T_0,F,T),\ \text{starts}(\neg F_1,T_1),\ T_0<T_1,\ T_1<T,\ F=F_1$$

and, since starts still has a "backward" rule (5), also to

$$\leftarrow \text{not\_clipped}(T_0,F,T),\ \text{initially}(F_1),\ T_0<0,\ 0<T,\ F=F_1$$

The latter IC can be dropped since $T_0<0$ is inconsistent with the general constraint $0<T_0$. The forward propagation rule induced by the first IC is

$$\mathbf{C}\text{not\_clipped}(T_0,F,T),\ \mathbf{C}\text{starts}(\neg F,T_1) \Rightarrow F\neq F_1\ ;\ F=F_1,(T_0>T_1\ ;\ T_1>T) \qquad (13)$$

---

plicitly propagated and we should therefore avoid having to forward propagate **C**not_p. not_p will thus be a backward predicate, used just to avoid violations of the IC ← not_p, p.

[8] Here, by constraints we mean predicates for which the Closed World Assumption does not hold. For example, the absence of $T1<T2$ from the constraint store does not entail $T1\geq T2$.

[9] Having both backward and forward versions of rule (6) does not lead to redundancies or loops. The role of the backward rule is to reduce a goal formulated in terms of starts(F,T) to assuming the abducible **C**happens(A,T) for an action A that initiates F. Then the forward rule propagates *all* other effects of A (not just the one that triggered the action application).

We have thus obtained the following set of rules

Backward goal reduction rules

$\mathrm{holds}(F,T) \Leftrightarrow \mathrm{starts}(F,T_0), T_0<T, \mathbf{C}\mathrm{not\_clipped}(T_0,F,T)$ (14)

$\mathrm{starts}(F,T) \Leftrightarrow \mathrm{initially}(F), T=0 \;;\; \mathrm{initiates}(A,F), \mathbf{C}\mathrm{happens}(A,T)$ (15)

$\mathrm{clipped}(T_0,F,T) \Leftrightarrow \mathrm{starts}(\neg F,T_1), T_0<T_1, T_1<T$ (16)

Forward propagation rules: (9)-(13).

We also have a general rule for all constraint predicates p:

$\mathbf{C}p(X_1) \setminus p(X_2) \Leftrightarrow X_1=X_2 \;;\; X_1\neq X_2, p(X_2)$ (17)

which is given a higher priority than the other rules and which tries to solve a goal $p(X_2)$ by reusing an already existing constraint $\mathbf{C}p(X_1)$ (propagated earlier by a forward rule). This rule also leaves the alternative of constructively achieving $p(X_2)$ open.

4.1 An improved CHR implementation

While the above approach avoids reachieving already solved *positive* goals, it doesn't avoid splitting when dealing with negative literals in *negative* goals (in our case 'not clipped' in negative goals originating from state constraints). An improved implementation would have to explicitly represent the disjunctive goals (involving clipped) before actually splitting them.

We shall represent partially activated state constraints as $\mathrm{ic}(\mathrm{Head} \leftarrow \mathrm{Body})$ (the initial state constraints have the form $\mathrm{ic}(\mathrm{fail} \leftarrow \mathrm{Body})$). Like in rule (11), such integrity constraints are (partially) activated by $\mathbf{C}\mathrm{starts}(F,T)$ constraints:

$\mathbf{C}\mathrm{starts}(F_1,T), \mathrm{ic}(\mathrm{Head} \leftarrow F_2, \mathrm{Body}) \Rightarrow F_1=_\exists F_2, \mathrm{ic}(\mathrm{Head} \;;\; \neg F_2|T \leftarrow \mathrm{Body})$ (18)
$\;;\; \sim(F_1=_\exists F_2)$

where $\neg F|T$ denotes the fact that the IC has been activated by a fluent F becoming true at time point T ($F_1=_\exists F_2$ is defined in Section 5.2 below).

The following rule inactivates an IC if a pair of its activating starts literals is clipped:

$\mathbf{C}\mathrm{starts}(\neg F,T), T_i<T, T<T_j \setminus \mathrm{ic}(\mathrm{Head};\neg F_i|T_i;\neg F_j|T_j \leftarrow \mathrm{Body}) \Leftrightarrow$ (19)
$F=F_i \;;\; \sim(F=F_i), \mathrm{ic}(\mathrm{Head};\neg F_i|T_i;\neg F_j|T_j \leftarrow \mathrm{Body})$

(Note that this rule has the form $\mathbf{C}\mathrm{clipped} \setminus \mathrm{ic} \Leftrightarrow \mathrm{true}$, or even
$\mathbf{C}\mathrm{clipped} \setminus (\mathrm{clipped} \;;\; \ldots \;;\; \mathrm{clipped}) \Leftrightarrow \mathrm{true}$.)

Finally, if an IC has been completely activated (without being inactivated by the previous rule, which has higher priority), then we should clip at least a pair of starts literals that activated it:

$\mathrm{ic}(\mathrm{Head} \leftarrow \mathrm{true}) \Rightarrow \mathrm{ic\_clip}(\mathrm{Head})$ (20)

Note that it is sufficient to clip a pair $(\neg F_i|T_i, \neg F_j|T_j)$ such that T_i has no known antecedent (w.r.t. the temporal order) in the set of activators, while T_j has no known successor:

```
ic_clip(Head) :-
    lower_bound(Head, ¬Fi|Ti), upper_bound(Head, ¬Fj|Tj), clipped(Ti, Fi, Tj).
```

We also have to deal with the possibility of ICs being activated by the initial state (while avoiding the forward propagation of the initial state):

$\mathrm{ic}(\mathrm{Head} \leftarrow \mathrm{Body}) \Rightarrow$ *forall* $\mathrm{Body} = F_1, \ldots, F_k$ *such that* $\mathrm{initially}(\mathrm{Body})$ (21)
$\mathrm{ic}(\neg F_1|0 \;;\; \ldots \;;\; \neg F_k|0 \;;\; \mathrm{Head} \leftarrow \mathrm{true})$.

The ICs propagated by these rules can of course be inactivated by the previous inactivation rule (19). (Note that in the improved approach rules (18)-(21) replace rules (11) and (12).)

The above rules have been directly implemented in the ECLiPSe as well as SICStus CHR environments. We have run tests comparing a simple partial order planner for independent fluents (similar to UCPOP and SNLP) with the planner described above and noticed no overheads (due to the treatment of dependent fluents) on planning problems with *independent* fluents. Since there are no other planners dealing with non-ground plans and dependent fluents, no standard benchmarks are currently available. However, we have successfully tested the planner on query planning problems in system integration, where dependent fluents occur naturally (as briefly described in the Introduction).

5 A general abductive procedure

In order to better clarify the relationship of our approach to existing abductive procedures, we present in the following a *general* abductive procedure that tackles the above-mentioned non-minimality problems by allowing a limited form of forward reasoning in addition to backward goal-directed reasoning. The procedure doesn't aim at improving the implementation from Section 4.1, its main role being to generalise the approach from the previous Sections. The transformation algorithm from Section 4, which *compiles* an ALP to CHRs is replaced by a general abductive algorithm that *interprets* the ALP directly. Of course, an *interpreter* is slower than a *compiled* procedure. However, besides providing a clarification of our approach, the general abductive procedure can also be used as an intermediate step for proving the soundness and completeness of our partial-order planning algorithm for dependent fluents. (The implementation from Section 4.1 above is more efficient due to its direct encoding in CHR, as opposed to using CHR just for *interpreting* positive and negative goals, as below. A number of problem and domain dependent decisions, such as declaring certain predicates to be "backward", also influence the efficiency of the implementation from Section 4.1. Let us stress the fact that these decisions are entirely domain and problem dependent and that they are not a drawback of our general mechanism.)

5.1 Open predicates

In Logic Programming, normal predicates are *closed*: their definition is assumed to be complete (Clark completion). On the other hand, abducibles in Abductive Logic Programming (ALP) are *completely open*, i.e. they can have any extension consistent with the integrity constraints. To formally deal with forward propagation rules in an abductive framework, we need to allow a generalization of abducibles, namely (partially) *open predicates*.

Unlike abducibles, open predicates can have definitions $p \leftarrow Body$, but these are not considered to be complete, since during the problem solving process we can add (by forward propagation) new abductive instances of p to these definitions. The definition of an open predicate therefore only partially constrains its extension.

In our CHR embedding of the abductive procedure we shall use two types of constraints, p and **C**p, for each open predicate p. While **C**p represent facts explicitly propagated (abduced), p refers to the *current closure* of the predicate p (i.e. the explicit definition of p *together* with the explicitly abduced literals **C**p). Thus, informally we have p = def(p) ∨ **C**p.

While propagating **C**p amounts to simply assuming p to hold (abduction), propagating p amounts to trying to prove p either by using its definition def(p), or by reusing an already abduced fact **C**p.[10] This distinction ensures that our CHR embedding conforms to the usual '*propertyhood view*' on integrity constraints:

Definition M(Δ) is a *generalized stable model* of the abductive logic program ⟨P,A,I⟩ for the abductive explanation $\Delta \subseteq A$ iff

(1) M(Δ) is a stable model of $P \cup \Delta$, and (2) $M(\Delta) \models I$.

The distinction between propagating **C**p and p respectively can be seen best in an example. When an action is applied, **C**happens(A,T), we have to propagate its effects Eff as well as its preconditions Pre. But while propagating the effects simply involves the propagation of the *constraint* **C**starts(Eff,T), propagating the preconditions should entail posting the *goal* holds(Pre,T),[11] which amounts to trying to achieve Pre either by using its definition (and thus applying another action having Pre as an effect), or by reusing an already achieved fact.

The use of *open predicates* allows mixing *forward propagation* of abduced predicates **C**p with *backward reasoning* using the closures p. Forward propagation can be implemented using CHR propagation rules, while backward reasoning involves unfolding predicates with their definitions. The definition def(p, Body) of a predicate p is obtained by Clark completion of its 'if' definitions. For each such predicate we will have an *unfolding rule* (a CHR simplification rule) p ⇔ def(p, Body) | Body, but also a CHR simpagation rule[12] for matching a goal p with an existing abduced fact **C**p:

$$\mathbf{C}p(X_1) \setminus p(X_2) \Leftrightarrow X_1=X_2 \;;\; X_1 \neq X_2, p(X_2).$$

This rule should be given a higher priority than the unfolding rule in order to avoid reachieving an already achieved goal. Combined with the forward propagation mechanism for **C**p, it deals with the first non-minimality problem mentioned in Section 2. Note that, for completeness, we are leaving open the possibility of achieving $p(X_2)$ using its definition or reusing other abduced facts.

Our treatment of open predicates p = def(p) ∨ **C**p is slightly different than the usual method [8] of dealing with (partially) open predicates p by introducing a new predicate name p' (similar to our **C**p) and adding the clause p ← p' to the definition of p:

---

[10] Abducibles (i.e. completely open predicates) p have no definition and are thus referred to as **C**p.

[11] This is done by the CHR rule (10) which corresponds to the ALP *integrity constraint* (7). Note that while program clauses propagate **C**p constraints, integrity constraints propagate goals p, in line with Lin and Reiter's observation [6] that state (integrity) constraints are usually intimately tied with the qualification problem.

[12] The rule is more complicated in practice, due to implementation details.

$\{p \leftarrow \mathrm{Def},\ p \leftarrow p'\}$. (**)

The difference is that whenever referring to p we are implicitly trying to prove p, either by using its definition def(p) or by reusing an already abduced fact **C**p, *but without allowing such a* **C**p *to be abduced in order to prove* p (whereas in (**) treating $p \leftarrow p'$ as a *program clause*[13] would allow p' to be abduced when trying to prove p). This is crucial for ensuring a correct distinction[14] between goals p and abducibles **C**p mentioned above (otherwise we would treat the propagation of action preconditions incorrectly). Without making this distinction, *we wouldn't even be able to refer to the current closure of* p.

5.2 The abductive procedure

The abductive procedure for open predicates given below is written using CHR rules.[15] We assume that conjunction ',' and disjunction ';' in positive goals are dealt with implicitly (disjunction being treated by splitting). Integrity constraints $\leftarrow G$ are represented as negative goals not(G).[16] The order of rules does matter: the first rule matching a newly introduced constraint will be activated. If it is a simplification rule, the subsequent rules will not get the chance to be executed.

In the rules below, we let p denote a predicate (possibly with variables). Multiple occurrences of p in a rule involves the unification of the corresponding literals. We also write p(X) (with X a tuple of variables) whenever we want to make the variables of p explicit.

Positive goals

[POS-ABD]	**C**$p(X_1) \backslash p(X_2) \Leftrightarrow X_1{=}X_2 \;;\; \sim(X_1{=}X_2), p(X_2)$.
[POS-UNF]	$p \Leftrightarrow \mathrm{def}(p, \mathrm{Body}) \mid \mathrm{Body}$.

Negative goals

[NEG-T,F]	not(true) $\Leftrightarrow$ fail.
	not(fail) $\Leftrightarrow$ true.
[NEG-UNF]	not(p, G) $\Rightarrow$ def_(p,Body) $\mid$ not(Body, G).
[NEG-ABD]	**C**$p(X_1)$ \ not($p(X_2)$, G) $\Leftrightarrow$ not($X_1{=}_\exists X_2$, G), not($p(X_2)$, $\sim(X_1{=}_\exists X_2)$, G)
[NEG-DISJ]	not((p_1; p_2), G) $\Leftrightarrow$ not(p_1, G), not(p_2, G).
[NEG-INACT]	**C**$p(X_1)$ \ not(G_1, not $p(X_2)$, G_2) $\Leftrightarrow$ no$\forall$vars(X_2) $\mid$ $X_1{=}X_2 \;;\; \sim(X_1{=}X_2)$, not(G_1,not $p(X_2)$, G_2)
[NEG-SPLIT]	not(not p(X), G) $\Leftrightarrow$ no$\forall$vars(X) $\mid$ p(X) ; not(p(X)), not(G).

[13] In fact, $p \leftarrow p'$ should be treated as an integrity constraint and not as a program clause.

[14] This distinction is essential only for *partially* open predicates and not for completely open predicates (abducibles).

[15] For lack of space, we omit the treatment of built-in constraints.

[16] 'not' is here a CHR constraint and should not be confused with the negation as failure operator used in logic programming.

The abductive procedure presented above is change-oriented, since a change in the abducible **C**p will trigger in rule [NEG-ABD] a matching negative goal, as in other abductive procedures. The main difference lies however in that **C**p can be *any* open predicate, not just a completely open one. Such open predicates can be targets of forward propagation rules: Body $\Rightarrow$ **C**p. If these propagation rules represent the forward direction of some program rules p $\leftarrow$ Body, then these program rules may be excluded when unfolding negative goals in [NEG-UNF]. There, def_(p, Body) returns the backward definitions of p, i.e. the Clark completion of the program rules for which no forward propagation rules have been written. For predicates p with no backward definition (for example for abducibles), def_(p, Body) returns Body = fail and the rule [NEG-UNF] propagates not(fail) i.e. true.

The main improvements of this abductive procedure consist in solving the problems of reachieving already achieved goals (mentioned in Section 2):

- in the case of positive goals by forward propagation
- in the case of negative literals in negative goals by inactivating the negative goals (using [NEG-INACT], whenever possible) before splitting them (using [NEG-SPLIT]). This also relies on forward propagation.

Negative goals can contain both universal ($\forall$) and existential ($\exists$) variables (the latter correspond to anonymous constants occurring in the constraint store). For variable tuples X_1 and X_2 we denote by $X_1 =_\exists X_2$ the set of equations obtained after eliminating (unifying away) the $\forall$ variables.[17] The no$\forall$vars(X) condition in the guard of [NEG-SPLIT] succeeds whenever the variable tuple X contains no universal variables. Its role is to avoid floundering. not(p(X)) in the second disjunct of the consequent of [NEG-SPLIT] implements a form of semantic splitting.

We have assumed conjunctions in the above algorithm to be ordered by a *selection function*. The selection function in a negative goal would typically prefer completely open predicates to other predicates and leave negative literals at the end. To avoid floundering, it would also try to choose only negative literals with no universal variables.

Our abductive procedure can be easily proved to be sound and complete. (The formal proof - which cannot be given here for lack of space - extends the standard proof for *SLDNFA* [2].)

For solving the planning problem with our general abductive procedure, we can simply use the general abductive logic program given by (3)-(8) together with the forward propagation rule (9) for starts. (Thus the rule (6) is "forward", the other ones, namely (3)-(5), being labeled "backward".)

[17] For example, if X_1=[Y,Z,Z], X_2=[A,B,C] with Y,Z $\forall$ variables and A,B,C $\exists$ variables, $X_1 =_\exists X_2$ is the set of equations {B=C} obtained after getting rid of Y and Z by Y=A, Z=B. We consider both cases $X_1 =_\exists X_2$ and $\sim(X_1 =_\exists X_2)$ in order to leave open the possibility that B$\neq$C.

6 Concluding remarks

Several related works, such as [9], use forward propagation for incrementally checking the consistency of deductive databases. Although they use forward propagation, the Kowalski-Sadri [10] and related algorithms are not appropriate for our purposes, since they only check the consistency of an update. The intermediate forwardly propagated facts are neither retained, nor reused after the check. These algorithms are therefore of no help to us for avoiding reachieving in a different way an already achieved goal.

On the other hand, the *Suspended Logic Programs* (SLPs - similar with Fung's IFF procedure) of [5], also allow a combination of backward goal-oriented reasoning with forward propagation of necessary properties. SLPs are very similar to CHRs in that insufficiently instantiated goals (that do not *match* any head of their iff definitions) are suspended. Thus, suspension is used as a mechanism for avoiding the combinatorial explosions that would be entailed by unfolding insufficiently instantiated goals. Instead of unfolding them, forward rules[18] are used to propagate the properties of such suspended goals in the hope of discovering any potential inconsistencies or for further instantiating the goal variables and thus allowing their unfolding.

Unfortunately, the propagated properties have to be unfolded as well, which may lead to a blow-up of the computation, unless special care is given to suspension control.[19] In our terminology, such SLP propagation rules propagate goals of the form p (which are subject to unfolding, leading to potential blow-ups), while we only propagate forward constraints of the form $\mathcal{C}$p (which are *not* subject to unfolding). Thus, SLPs represent a more general architecture (very much like CHRs), while we are developing a more specific abductive procedure, being more concerned with dealing with the non-minimalities in existing abductive procedures.

As far as we know, our planner is the first sound and complete partial-order planner able to deal with dependent fluents and non-ground plans. The prototype CHR implementation has been successfully tested as query planner of the mediator in the framework of system integration.

Acknowledgments

The work presented here has been partially supported by the European Community project SILK (IST-11135).

18 Similar to CHR propagation rules.

19 In fact, Wetzel started studying the use of deletion rules [11] for alleviating the blow-up of propagated properties.

References

[1] Denecker M., Missiaen L., Bruynooghe M. Temporal reasoning with abductive event calculus, ECAI-92, 384-388.

[2] Denecker M., DeSchreye D. SLDNFA: an abductive procedure for abductive logic programs, J.Logic Programming 34(2),111-167, 1998.

[3] Fruewirth T. Constraint Handling Rules, in Podelski A. (ed) Constraint Programming: Basic and Trends, LNCS 910, 90-107, 1995.

[4] Kakas A., Michael A., Mourlas C. ACLP: a case for non-monotonic reasoning, Proc. NM'98.

[5] Kowalski R., Toni F., Wetzel G. Executing suspended logic programs, Fundamenta Informaticae 34 (1998), 1-22.

[6] Lin F., Reiter R. State constraints revisited, JLC4(5), 655-678.

[7] Shanahan M. An abductive event calculus planner, JLP44 (2000)

[8] Kakas A., Kowalski R., Toni F. The role of abduction in logic programming, Handbook of logic in AI and LP 5, OUP1998, 235-324.

[9] Kowalski R., Sadri F., Soper P. Integrity checking in deductive databases, VLDB'97.

[10] Bylander T., Allemang D., Tanner M.C., Josephson J.R. The computational complexity of abduction, AIJ 49(1-3), pp. 25-60, 1991.

[11] Wetzel G. Using integrity constraints as deletion rules, Proc. DYNAMICS'97, 147-161, 1997.