CHICA, an abductive planning system based on Event Calculus

Lode Missiaen*
Maurice Bruynooghe†
Marc Denecker
Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001 Heverlee
Belgium
missiaen@stc.nato.int

26 April 1994

Abstract

This article presents the theory and implementation of an Artificial Intelligence planner, CHICA. CHICA is a non-linear, domain independent planner based on techniques of computational logic. The representation language of the planner is Horn clause logic which is used to model *event calculus*, a logical theory of changing properties over time. The reasoning component is an abductive extension of SLDNF resolution for generating assumptions to prove a given goal. In event calculus, this procedure generates a plan of events and temporal relations necessary to prove the planning goal. CHICA uses domain contraints and techniques from contraint logic programming to efficiently implement inequality, as well as a specialized module to evaluate temporal relations. CHICA's generic search algorithm lets the implementor of a planning domain define a particular search strategy and specify domain heuristics to prune the search space. CHICA has solved a number of planning problems successfully: multiple robot block world problems, the assembly of a flashlight, and a room decoration problem. Extensions to classical AI-planning can be solved within the same framework, such as plan execution and replanning.

keywords: planning, logic programming, event calculus

*Senior Scientist at SHAPE Technical Centre, The Hague, The Netherlands
†Research Associate of the National Fund for Scientific Research, Belgium

1 Introduction

AI planning is the subfield of *Artificial Intelligence (AI)* that investigates the automation of reasoning about actions. Planning is a necessary element of intelligent behavior, and its effective implementation is important for future applications in manufacturing processes, robotics, automated navigation, management strategies, and other real-world applications.

Actions change the properties of a problem domain, for example the action *taking a shower* causes one to become *wet*. Given a description of possible actions and how each of these actions changes the state of affairs, a planner tries to find which actions, together with their ordering in time, achieve a given *goal* starting from an initial state of the world. This article restricts the planning problem to *domain-independent preplanning*; the techniques used to generate a plan are independent of the problem domain and the plan is constructed prior to its execution. The planner constructs the plan in a *closed world* in which all relevant actions and properties are known. During its execution, no events external to the plan can change this world.

We summarize the results of a study on the use of computational logic to build a practical planning system, CHICA. CHICA's logic representation language has a clear semantics and is more expressive than the traditional add-and-delete list representation of actions. Plan modification operations are achieved by an abductive extension of logic programming, which enables CHICA to use the deduction engine of logic programming and to apply constraint logic programming techniques. CHICA's search algorithm is defined in terms of the search space of a logic program which enables us to define domain specific heuristics in terms of computation, selection, and pruning rules. The logical framework of CHICA provides a good basis to determine precisely the effect of heuristics on the search space.

Section 2 explains the *abductive proof procedure*, the proof procedure of logic programming extended with an inference rule called *abduction*. Abduction generates possible hypotheses to prove a given goal from a theory. Section 3 defines CHICA's logical theory of planning, which is based on *event calculus*. This theory defines how a property changes its truth value when an event happens. In Section 4, we apply the abductive proof procedure to the event calculus theory, thus obtaining a reasoning component for planning. Section 5 describes CHICA's implementation of this reasoning component, and Section 6 describes CHICA's search control mechanism. Section 7 compares CHICA with related work and with other domain independent planning systems. Section 8 describes CHICA's application to three planning problems. In Section 9, suggestions are made for extensions and future research.

2 Abductive Proof Procedure

2.1 SLDA

SLDA is an abductive extension of SLD, Selection rule driven Linear resolution for Definite clauses. In logic programming, the SLD proof procedure proves that a given goal G follows from a given logical theory T: $T \vdash G$. The SLD proof procedure uses a rule of inference called *resolution* which is based on *modus ponens* augmented with unification:

from $p \leftarrow q$ **and** q **infer** p

SLDA is an extension of the SLD proof procedure in that it computes a set of unit clauses Δ restricted to a set of abducible predicates $\mathcal{A}$ that must be assumed in order to prove a given ground goal G from a theory T:[1] $T + \Delta \vdash G$. SLDA is based on the abduction inference rule; given a rule $p \leftarrow q$ and a fact p, abduction generates an explanation for p by stating the fact q.

from $p \leftarrow q$ **and** p **infer** q

This is not a sound inference rule, however, because q does not logically follow from $p \leftarrow q$ and p. Therefore, q must be seen as a hypothetical explanation. If the implication $p \leftarrow q$ corresponds to the notion of *causality* — which is an intuitive concept — then abduction will generate plausible explanations.

(1.1) faulty(L) $\leftarrow$ lamp(L), *broken(L).*
(1.2) faulty(L) $\leftarrow$ lamp(L), current_break(L).
(1.3) current_break(L) $\leftarrow$ *fuse(L,F), melted_fuse(F).*
(1.4) current_break(L) $\leftarrow$ *general_power_failure.*

(1.5) lamp(a).
(1.6) lamp(b).
} facts

Program 1 Diagnosis of faulty lamp

Consider the theory T about the diagnosis of a faulty lamp (Program 1). Suppose we want an abductive proof for the goal $G =$ *faulty(a)*. Let the set of abducible predicates, $\mathcal{A}$, be equal to the set of predicates that have no definition in T: $\mathcal{A} = \{$*broken/1, fuse/2, melted_fuse/1, general_power_failure/0*$\}$[2]. There are three possible extensions for T corresponding to three possible explanations why lamp a is faulty:

$\Delta_1 = \{$ broken(a) $\}$
$\Delta_2 = \{$ fuse(a, α), melted_fuse(α) $\}$
$\Delta_3 = \{$ general_power_failure $\}$

In other words, if we extend T with the facts of one of the sets Δ_i, we can prove *faulty(a)*. In Δ_2, α is a *skolem constant* that represents an unknown element of the problem domain. Skolem constants are designated by lowercase Greek letters.

SLDA is an extension of existing residue procedures in which Δ is the residue left at the dead ends of a resolution proof [Finger and Genesereth 85, Cox and Pietrzykowski 86]. In our scheme, abducibles may be selected by introducing skolems for variables by *extended unification*. These abducible goals can be not only abduced but also resolved with previously abduced facts. SLDA constructs a refutation

$$< G^0, \Delta^0 >, \ldots, < G^i, \Delta^i >, \ldots, < G^n, \Delta^n >$$

[1] We assume a ground goal for simplicity. In the remainder of this article, all of the results stated for ground goals can be reformulated for non-ground goals by introducing an answer substitution.

[2] Predicates and compound terms are often represented by their name and the number of arguments, separated by a slash.

In this refutation, G^i is an intermediate goal clause, Δ^i is a set of abducible unit clauses, G^0 is equal to the original ground goal G, Δ^0 is the empty set, G^n is the empty clause ($\Box$), and Δ^n is the solution set of abducibles. $< G^{i+1}, \Delta^{i+1} >$ is obtained from $< G^i, \Delta^i >$ as follows: Suppose G^i is the conjunction of literals $g_1, \ldots, g_k, \ldots, g_m$. One literal g_k is selected, and either a *resolution step* or an *abduction step* is performed.

In a resolution step, g_k is resolved with one of the clauses in $T + \Delta^i$ using *extended unification* which binds skolems as well as variables. Let $h \leftarrow l_1, \ldots, l_u$ be a variant of the selected clause that does not contain any variables used in the refutation until $< G^i, \Delta^i >$. Extended unification constructs a most general unifier (mgu) of the literals g_k and h, $\theta\sigma$, with σ a variable substitution and θ a (non-variable) skolem substitution. Thus,

$$G^{i+1} = \theta\sigma(g_1, \ldots, g_{k-1}, l_1, \ldots, l_u, g_{k+1}, \ldots)$$
$$\Delta^{i+1} = \theta(\Delta^i)$$

Extended unification of *fuse(a,α)* and *fuse(X,f(X))* results in a *mgu* $\theta\sigma$, with $\sigma = \{X/a\}$ and $\theta = \{\alpha/f(a)\}$. The *mgu* of *melted_fuse(α)* and *melted_fuse(f(Y))* is $\sigma = \{Y/\beta\}$ and $\theta = \{\alpha/f(\beta)\}$. β is a new skolem constant bound to the variable Y which avoids placing Y in θ.

An abduction step can be applied if g_k is abducible. Let σ^{i+1} be the variable substitution replacing all of the variables in g_k with new skolem constants. σ^{i+1} is called the *skolemizing substitution* of g_k. Then,

$$G^{i+1} = \sigma^{i+1}(g_1, \ldots, g_{k-1}, g_{k+1}, \ldots, g_m)$$
$$\Delta^{i+1} = \Delta^i + \sigma^{i+1}(g_k)$$

Figure 1 is the abductive proof tree of the lamp diagnosis problem, representing the top-down search space of all possible abductive refutations for a given computation rule or literal selection rule. The selected literals are underlined. The accumulated set of abducibles, Δ, is represented next to the derivation. Each resolution step is indicated with the rule number applied in Program 1. Abduction steps are indicated with roman numbers.

The given goal *faulty(a)* can be resolved by either Rule (1.1) or Rule (1.2). Resolution with Rule (1.1) results in *lamp(a), broken(a)*. The goal *lamp(a)* is resolved with Fact (1.5). The remaining goal, *broken(a)*, cannot be resolved with a rule in Program 1 or with a fact in Δ. However, *broken/1* is abducible, and therefore, *broken(a)* is solved through an abduction step, (i), adding *broken(a)* to Δ. At this point, all of the goals are solved and Δ={broken(a)}.

The subtree that originates from the resolution of *faulty(a)* with Rule (1.2) can be constructed similarly. It contains two other refutations with corresponding solutions for Δ. Notice that in abduction step (ii) the abduced literal *fuse(a,F)* is skolemized by replacing F with the new skolem constant α: $\sigma = \{F/\alpha\}$.

2.2 SLDNFA

SLDNFA is general abductive proof procedure for normal programs, which contain rules with negative conditions [Denecker Missiaen and B. 92]. Such proof procedures already existed before SLDNFA for the propositional case and for safe computation rules that only select ground abducible atoms [Esghi and Kowalski 89, Kakas and Mancarella 90]. However, in

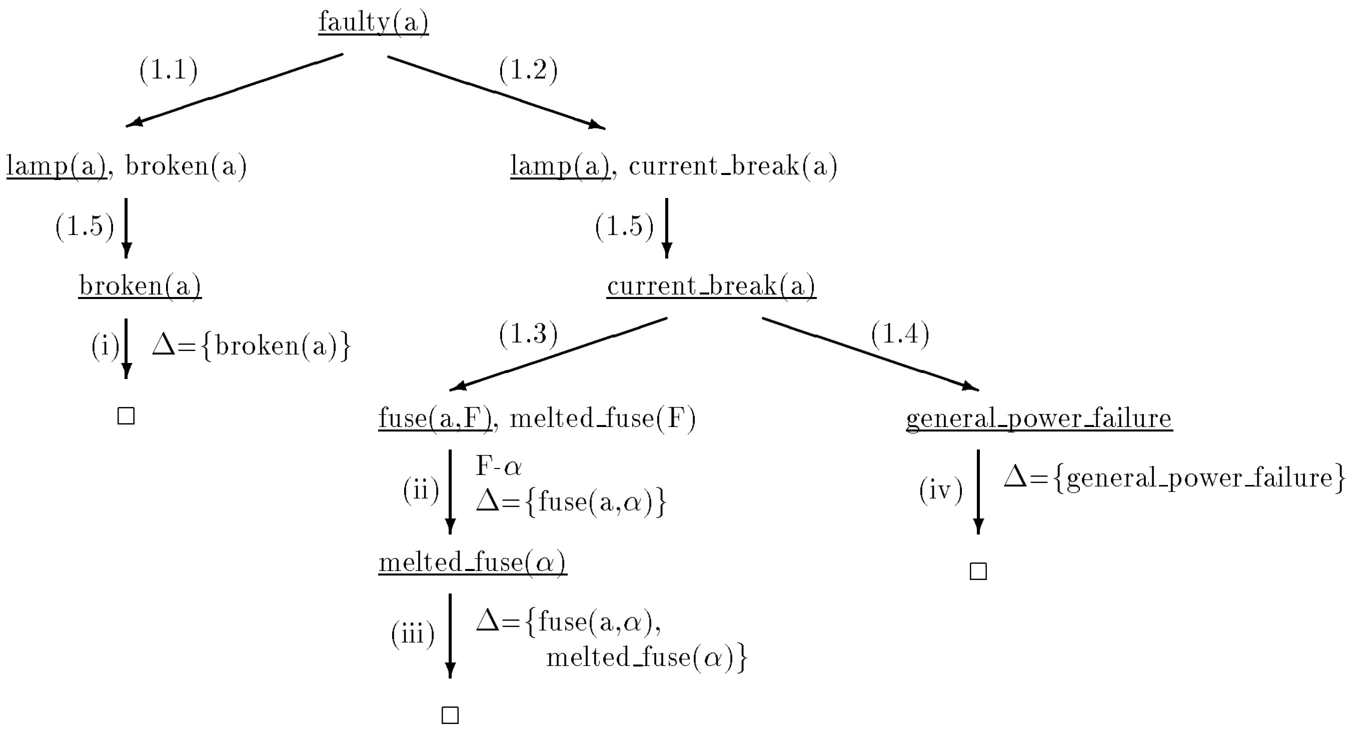


Figure 1 Proof tree of lamp diagnosis problem

planning, non-ground literals must be abduced through a skolemizing substitution in order to generate new events. SLDNFA treats abducible goals distinctly depending on their depth of execution. At an event depth, where the goals are positive, an abducible goal will be resolved or abduced. At an odd depth, where the goals are negative, the goals are only resolved. The failure tree of a negative goal must be reconsidered every time a new abducible is added to Δ. SLDNFA avoids recomputation by interleaving the computation of this failure tree with the construction of Δ. We do not present SLDNFA in detail here because CHICA uses a specialized version of SLDNFA for event calculus, explained in Section 4.

3 Event Calculus for Planning

3.1 Event Calculus

Kowalski and Sergot presented *event calculus* in 1986 as a logical framework for the representation of, and the reasoning about, time [Kowalski and Sergot 86]. It was originally used as a mechanism for updating databases and understanding natural language. Since then, event calculus has been presented in a variety of other ways [Sadri 87, Shanahan 89, Shanahan 90, Borillo and Gaume 90, Sripada 91, Missiaen 91b].

The event calculus is based on *events* which initiate and terminate properties. For example, the event *taking a shower* would make the property *one's hair is wet* true. Events are assumed to be atomic; no other events can happen during the execution of an event. The occurrence of events allows axioms to deduce the existence of time periods during which properties are true. This is opposed to the state-based approach of *situation calculus* which is used by most planning systems. In situation calculus, actions are viewed as state transformers, a state being the set of properties true at a particular time [McCarthy and Hayes 69]. The event calculus is more advantageous than situation calculus because events do not need to be totally ordered. This alleviates the computational aspect of the *frame problem* in situation calculus. Moreover, event calculus can express and calculate the truth value of *derived properties* which would lead to inconsistencies in situation calculus [Missiaen 89].

3.2 Event Calculus for Planning

The event calculus for planning is a logical theory of time for representing planning problems. This theory consists of three parts: the domain independent theory, the definitions of the problem domain, and the description of the planning problem. An event calculus plan is an ordering of events to achieve a given planning goal.

3.2.1 Domain independent theory

The domain independent theory consists of the axioms of Program 2. It corresponds to the *Modal Truth Criterion* in situation calculus [Chapman 87].

(2.1) holds_at(P, T) $\leftarrow$
    happens(E), initiates(E, P), succeeds(E),
    E $\prec$ T, maintain(E, P, T).

{*Persistence rules*}
(2.2) maintain(E, P, T) $\leftarrow$ **not** clipped(E, P, T).
(2.3) clipped(E, P, T) $\leftarrow$ happens(C), terminates(C, P), **not** out(C, E, T).
(2.4) out(C, E, T) $\leftarrow$ (T = C ; T $\prec$ C ; C $\prec$ E)[a].

{*Maintenance rules*}
(2.5) maintain(E, P, T) $\leftarrow$ **not** m_clipped(E, P, T).
(2.6) m_clipped(E, P, T) $\leftarrow$
    happens(C), terminates(C, P),
    E $\prec$ C, C $\prec$ T,[b]
    not white_knight(C, T, P).
(2.7) white_knight(C, T, P) $\leftarrow$
    happens(W), initiates(W, P), succeeds(W),
    C $\prec$ W, W $\prec$ T, maintain(W, P, T).

Program 2 Domain independent theory

[a] *(p ; q ; r)* should be read as *p or q or r*.
[b] The case $E = C$ does not need to be considered because E is always an initiating event for the property P.

Time points are ordered by the precedence predicate $\prec/2$. The predicate *holds_at(P,T)*, defined by Rule (2.1), defines when a property P holds true at some time point T in terms of past events. The predicate *happens(E)* represents the fact that the event E occurs; the occurrence of an event has no duration, and therefore we also use E to represent the time point at which E occurs. The predicate *initiates(E,P)* states that the event E initiates the property P. The predicate *succeeds(E)* states that the preconditions of the event E are satisfied. The predicate $E \prec T$ states that E must precede T; as a result, E initiates P to be true after E. The last condition of Rule (2.1) is *maintain(E,P,T)* which states that the property P holds at a sub-interval of $[E, T]$, excluding E, but including time point T. The predicate *maintain(E,P,T)* has two definitions: (2.2) and (2.5).

Rules (2.2) to (2.4) are called *persistence rules*. They define that *maintain(E,P,T)* is true in the absence of an event C in the interval $[E, T]$ that terminates P. The existence of such an event is defined by the predicate *clipped(E,P,T)*.

Rules (2.5) to (2.7) are called *maintenance rules*; for every event C that terminates P, there must be an event W which restores P. P must be temporally ordered between C and T, and P must be maintained over the interval $[W, T]$.[3] The maintenance rules are redundant to the persistence rules because solutions obtained using the maintenance rules can always be obtained through backtracking using the persistence rules only. However, the maintenance rules are precisely added to avoid backtracking (Section 4.2.4).

3.2.2 Domain theory

A problem domain is a closed universe with a limited number of actions and properties. The theory of a problem domain defines how actions affect properties. It also defines the preconditions of actions, the properties that must hold true to execute the action successfully. In event calculus, an event E is an instance of some action A, expressed by the predicate *act(E,A)*.

The domain component of the event calculus theory uses the predicate *succeeds/1* to define the preconditions of the actions. For example, in Program 3, Rule (3.1) states *X must be clasped* and *Y must be clear* when *block X is put on location Y*. The predicates *initiates/2* and *terminates/2* define the effects of these actions. In Rule (3.3), *putting X on Y* initiates the property *X is on Y*, and in Rule (3.8), this action terminates the property *block Y is clear*.

3.2.3 Problem description and goal

The problem description consists of the objects in the problem and the initial situation. A special event, *start*, is used as the initial situation. This event *start* precedes all other time points and initiates the properties that are initially true. For example, facts (4.1) to (4.7) define the initial situation of Program 4, Sussman's anomaly block world problem.[4] Facts (4.8) to (4.12) define the objects: the blocks and locations. The denial (4.13) presents the goal G of the problem.

[3] In Chapman's modal truth criterion, E is called an *establisher*, C is called a *clobberer*, and W is called a *white knight* [Chapman 87].

[4] The blocks world example of Program 4 is called *Sussman's anomaly* because Sussman's planner HACKER could not solve it [Sussman 73].

```
(3.1)  succeeds(E) ←
             act(E, put(X,Y)), block(X), location(Y), X≠Y,
             holds_at(clasped(X), E), holds_at(clear(Y), E).
(3.2)  succeeds(E) ←
             act(E, pick(X)), block(X),
             holds_at(free_robot, E), holds_at(clear(X), E).

(3.3)  initiates(E, on(X,Y)) ← act(E, put(X,Y)).
(3.4)  initiates(E, clear(X)) ←  act(E, pick(Y)), holds_at(on(Y,X), E), block(X).
(3.5)  initiates(E, clear(X)) ← act(E, put(X,Y)).
(3.6)  initiates(E, clasped(X)) ← act(E, pick(X)).
(3.7)  initiates(E, free_robot) ← act(E, put(X,Y)).

(3.8)  terminates(E, clear(Y)) ← act(E, put(X,Y)), block(Y).
(3.9)  terminates(E, clear(X)) ← act(E, pick(X)).
(3.10) terminates(E, on(X,Y)) ← act(E, pick(X)).
(3.11) terminates(E, clasped(X)) ← act(E, put(X,Y)).
(3.12) terminates(E, free_robot) ← act(E, pick(X)).
```

Program 3 Domain theory of the blocks world

3.2.4 Event calculus' plan

An event calculus' plan specifies its actions ordered in time. A plan can be defined by facts for the predicates *happens/1* and *act/2*, and the precedence predicate $\prec/2$. Program 5 is an example of such a plan. Program 5 is a partially ordered plan because the precedence relation between the time points is not completely defined. Any total ordering yields a linear plan.

Given an event calculus theory T and a (ground) planning goal G, a plan P is a *solution* if and only if

1. $T + P \vdash G$, and
2. $\forall E_i : happens(E_i) \in P \Rightarrow \; T + P \vdash succeeds(E_i)$

The first condition states that the given goal G can be proven from the event calculus theory extended with the facts from P. The second condition states that the preconditions of the events of the plan must be satisfied. A solution plan P is a *valid* plan if all the linearizations of P are also solutions. In Section 4.3, we shall see that a solution plan is not always a valid plan.

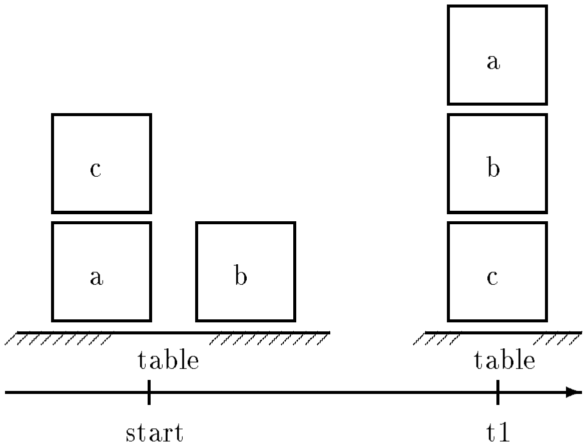
4 Abduction in Event Calculus

This section adapts SLDA of Section 2 to the event calculus theory of Section 3 to obtain a planning mechanism.

(4.1) initiates(start, on(c,a)).
(4.2) initiates(start, on(a,table)).
(4.3) initiates(start, on(b,table)).
(4.4) initiates(start, clear(c)).
(4.5) initiates(start, clear(b)).
(4.6) initiates(start, clear(table)).
(4.7) initiates(start, free_robot).

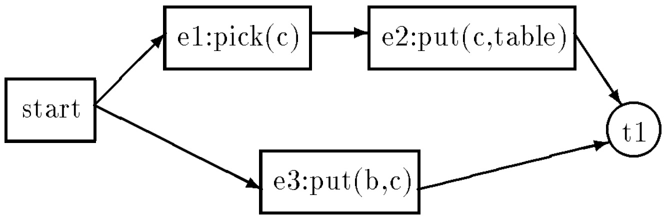
(4.8) block(a).
(4.9) block(b).
(4.10) block(c).
(4.11) location(table).
(4.12) location(B) ← block(B).

(4.13) ← holds_at(on(b,c), t1),
holds_at(on(c,table), t1),
holds_at(on(a,b), t1).



Program 4 Sussman anomaly problem

(5.1) happens(e1).
(5.2) act(e1, pick(c)).
(5.3) happens(e2).
(5.4) act(e2, put(c,table)).
(5.5) happens(e3).
(5.6) act(e3, put(b,c)).
(5.7) $e1 \prec e2$.
(5.8) $e2 \prec t1$.
(5.9) $e3 \prec t1$.



Program 5 Partial plan

4.1 Planning as abduction

A solution plan P in event calculus must satisfy $T + P \vdash G$. Given T and G, any planning problem consists of finding one or more valid plans P that satisfy this requirement. The implication $T + P \vdash G$ matches $T + \Delta \vdash G$ of the abduction framework of Section 2. Hence, planning can be reduced to an abductive derivation by choosing abducible predicates that constitute a plan in event calculus: $\mathcal{A} = \{happens/1, act/2, \prec/2\}$.

4.2 Specializations of the abductive proof procedure

SLDA must be modified for event calculus to ensure correctness of plans, to improve efficiency and to deal with negative goals in event calculus. Negative goals are solved by an abductive extension of negation by finite failure, a restricted version of the general mechanism used by

SLDNFA.

4.2.1 The event skolems and temporal relations

The abducible temporal relation $\prec/2$ is transitive, asymmetric and irreflexive. These properties are not represented explicitly as rules in an event calculus theory because they could not be used efficiently. Therefore, abduced literals of $\prec/2$ are collected in a separate set $\Delta t \subset \Delta$. Every time a relation $t1 \prec t2$ is added to Δt, we compute the transitive closure of Δt. If $t2 \prec t1$ already belongs to Δt, then the abduction of $t1 \prec t2$ fails.

The facts for *happens/1* and *act/2* are collected in Δp, so $\Delta = \Delta p + \Delta t$. Abduction of *happens(E)* will skolemize the event variable E as $E - \epsilon_1$, with ϵ_1 representing an *event skolem*. Event skolems represent unique constants for the events that happen in the plan. Other skolem constants that do not represent events are called *object skolem constants* (such as blocks and locations). During abductive refutation, extended unification can bind skolems to terms (Section 2.1). Event skolems are prevented from binding in order to avoid multiple actions in Δp becoming associated with the same event.

4.2.2 Inequalities and the domain constraints

SLDA must be extended to handle inequalities $\neq$ in the body of the domain rules such as Rule (3.1) in Program 3. We use the *free equality theory* to determine the truth value of a literal $x \neq y$ [Clark 78]. This set of axioms constrains the possible interpretations of $x = y$ and $x \neq y$. If the free equality theory cannot determine the truth value of $x \neq y$, $x \neq y$ will be solved as an abducible predicate; $x \neq y$ will be skolemized and added to a set Δs. Every time a skolem is bound, some of the inequalities in Δs must be rechecked.

In a problem description, we specify domain constraints on variables, replacing untyped variables with variables ranging over a finite set of possible values called a *domain* [Van Hentenryck 89]. For example, in the block world problem of Program 4, we can replace the facts (4.8) to (4.10) with a single fact stating that the argument of *block/1* ranges over the set $\{a, b, c\}$. In the same way we can state that the argument of *location/1* ranges over the domain $\{a, b, c, table\}$. Predicates such as *block/1* and *location/1* define domain variables and are called *domain predicates*. To solve a literal such as *block(B)*, we skolemize its argument as $B - \beta$ and add the constraint $\beta \in \{a, b, c\}$ to Δs. As a result, a deterministic step replaces the nondeterminism of a domain predicate at the resolvent level, reducing branching considerably. In addition, equality and inequality constraints are used as soon as possible to reduce the set of values that can be given to skolems. Suppose that α is constrained to the domain $\{a, b, c\}$ and β to the domain $\{a, b, d\}$, then the equation $\alpha = \beta$ will assign the intersection of the domain of α and β, $\{a, b\}$, as their new domain. If the intersection is empty, $\alpha = \beta$ will fail; if it is a singleton, both α and β will be bound to the singleton value. Propagation reduces the domain of a skolem in an inequality as soon as the other skolem is bound. For example, if $\alpha \neq \gamma \in \Delta s$, α is constrained to the domain $\{a, b\}$, γ gets bound to b during a refutation, then the inequality $\alpha \neq \gamma$ is propagated, $\alpha \neq b$, and $\alpha \in \{a, b\}$ reduces to $\alpha \in \{a\}$ resulting in binding α to a.

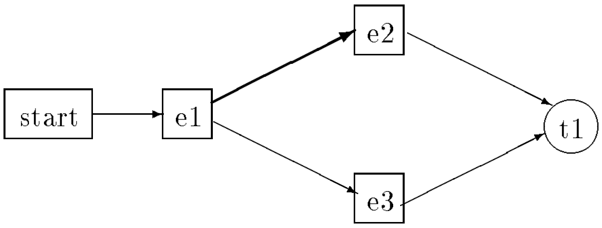


Figure 2 Abduction of $e1 \prec e3$

4.2.3 Persistence constraints Δc

In the persistence rules of Program 2, there are two negative literals: *not clipped(E,P,T)* and *not out(C,E,T)*. To prove *not clipped(E,P,T)*, the complete search space of *clipped(E,P,T)* must be explored. If we want *clipped(E,P,T)* to fail, abduction steps should not be performed in the body of Rule (2.3) in order to prove the literals *happens(C)* and *terminates(C,P)*. Instead, they must be executed deductively, treating skolems as ordinary constants. On the other hand, to prove the failure of *not out(C,E,T)*, we want the call *out(C,E,T)* to succeed. The literal *out(C,E,T)* is a positive goal and therefore, it must be executed abductively so that abduction steps can add precedence relations to Δt via Rule (2.4). For example, if a refutation arrived at the plan of Program 5 and the literal to be solved were *maintain(start,clear(c),e1)*, then the execution of the persistence rules, (2.2) to (2.4), would prove *happens(e3)* and *terminates(e3,clear(c))* deductively, but *not out(start,e3,e1)* would fail through the abduction of $e1 \prec e3$. This would result in the new plan presented in Figure 2.

The success of the call *not clipped(E,P,T)* is *non-monotonic* in an abductive refutation; the abduction of new events and the binding of skolems can make a previously failing goal *clipped(E,P,T)* succeed. Therefore, the proven goals *not clipped(E,P,T)* are accumulated in a set Δc, and every time a new event is abduced or a skolem gets bound, the negative goals in Δc must be re-executed, possibly adding new time relations to restore their truth. The goals in Δc are also called constraints. Checking every constraint in Δc would be computationally expensive. The actual implementation uses a method based on locality to determine the constraints in Δc selectively that could be affected by adding a new event [Missiaen 91b]. Techniques for checking integrity constraints in deductive databases could be used for this same purpose [Sadri and Kowalski 88, Bry Manthey and M. 90].

4.2.4 Maintenance mechanism

If a goal in Δc can no longer be proven via the persistence rules, then the abductive refutation would fail, and backtracking would return control to a previous choice point where an alternative branch would be selected. Experiments show that such backtracking is inefficient because it removes abduction steps in the failing refutation that have to be recomputed in the alternative refutation. Therefore, to avoid backtracking, we replace the failing constraint in Δc with a new constraint and modify the plan appropriately. This

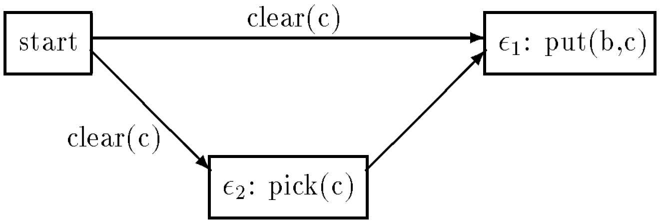


Figure 3 Violated **not** clipped(start,clear(c),ϵ_1)

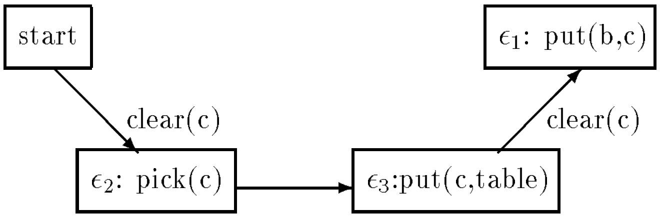


Figure 4 Maintenance mechanism

maintenance operation is performed using maintenance rules (2.5) to (2.7). It corresponds to the *white knight* mechanism of the planner TWEAK [Chapman 87]. In Rule (2.6), the literals *happens(C)*, *terminates(C, P)*, $E \prec C$, and $C \prec T$ are executed deductively with skolems treated as constants, so no abduction can happen. The literals of the body of Rule (2.7) are called at the positive level of the nested negative call, *not white_knight(C,T,P)*. These literals can be refuted with the abductive proof procedure, abducing new events and new time relations.

For example, if we arrive at the plan of Figure 3 with $\Delta c = \{$*not clipped(start, clear(c), ϵ_2), not clipped(start, clear(c), ϵ_1)*$\}$, then the constraint *not clipped(start,clear(c),ϵ_1)* would be violated and could not be satisfied with the persistence rules. The maintenance step would remove the constraint *not clipped(start, clear(c), ϵ_1)* from Δc and solve the goal *maintain(start, clear(c), ϵ_1)* using the maintenance rules, (2.5) to (2.7). Rule (2.6) would find that ϵ_2 was a terminating event of *clear(c)* inside interval $[start, \epsilon_1]$. Rule (2.7) would abduce a new event ϵ_3, *act(ϵ_3,put(c,table))*, initiating *clear(c)*, and the precedence relations $\epsilon_2 \prec \epsilon_3$ and $\epsilon_3 \prec \epsilon_1$. This would result in the plan of Figure 4. In general, a maintenance step must perform an additional operation on Δt as follows. Together with the removal of *not clipped(e,p,t)* from Δc, all of the time relations abduced in Rule (2.4) during checking this constraint but not used elsewhere, must be removed from Δt. In other words, all of the hypotheses that were used solely to contruct the failure tree of *not clipped(e,p,t)* need to be removed from Δt.

4.3 Correctness and completeness problems

Event calculus logic is an expressive representation language for planning problems. Event calculus can represent context-dependent and indirect effects. Chapman has proven that non-linear planning with such effects is undecidable [Chapman 87]. All non-linear planners that allow for such effects are incomplete and generally incorrect, and CHICA is no exception. In this section, we exemplify the correctness and completeness problems, formulate practical solutions, and relate the problems back to correctness and completeness of SLDNFA.

4.3.1 Correctness

In the simple planning problem of Program 6, the *terminates* rules define the context dependent effects: $e1$ terminates r if q holds, and $e2$ terminates r if p holds. The abductive proof procedure will find the solution plan drawn in Figure 5. Indeed, at $e1$, we cannot prove that q holds, and, therefore, r must remain true until $t1$. A similar argument applies to $e2$. However, this plan is invalid since none of its linearizations are solutions, as shown in Figure 5. In the first linearization, $e1$ is put before $e2$ so the condition p under which $e2$ terminates r becomes true. The argument is symmetric in the second linearization. Therefore, the solution plan of Figure 5 is considered incorrect.

```
initiates(start, r).                              succeeds(e1).
initiates(e1, p).                                 succeeds(e2).
initiates(e2, q).
terminates(e1,r) ← holds_at(q,e1).
terminates(e2,r) ← holds_at(p,e2).

← holds_at(p, t1), holds_at(q, t1), holds_at(r, t1).
```

Program 6 Correctness problem

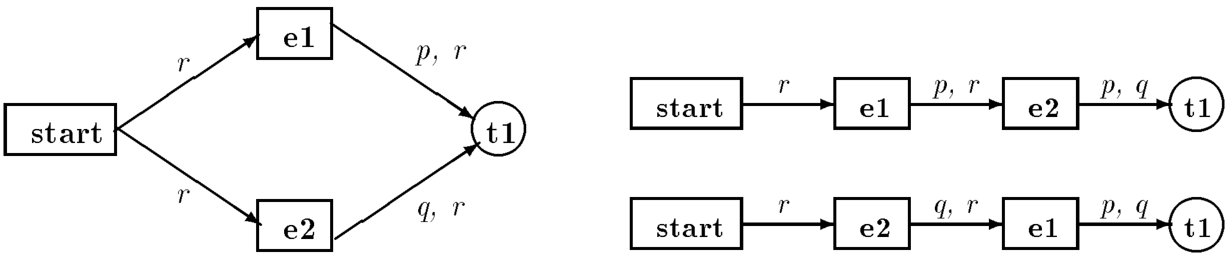


Figure 5 Solution plan for Program 6 and two possible linearizations

4.3.2 Completeness

For the simple planning problem in Program 7, the abductive proof procedure would find the failing intermediate solution of Figure 6. Since q holds, $e1$ terminates r which is true initially. In the given problem, there is no event that could initiate r, so the abductive proof procedure would terminate with no solution plan. However, the failing plan in Figure 6 could be turned into a solution plan with the addition of the event $e2$ which would terminate q, the condition under which $e1$ would terminate r. The abduction of such an event would happen inside a nested call of *terminates(C,P)* in Rule (2.3). But in the specialized abductive proof procedure, this could never happen because the literal *terminates(C,P)* in Rule (2.3) is executed deductively, preventing abductions.

```
initiates(start, r).                    succeeds(e1).
initiates(start, q).                    succeeds(e2).
initiates(e1, p).
terminates(e1,r) ← holds_at(q,e1).
terminates(e2,q).

← holds_at(p, t1), holds_at(r, t1).
```

Program 7 Completeness problem

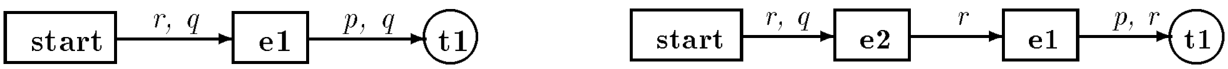


Figure 6 Incorrect plan and solution plan for Program 7

4.3.3 Solutions to completeness and correctness

The previous examples show that correctness and completeness problems occur in the case of context dependent terminating events. However, we do not exclude such events because they can express interesting problems.

In cases where correctness or completeness poses a problem, one of the following methods can be used:

Linearization The problem of correctness can always be solved by considering all possible linearizations of every intermediate plan. However, this method is impractical for large planning problems because the number of linearizations of a partial ordering grows exponentially with the number of time points in that ordering. In CHICA, only subsets of events are considered for linearization. Using a method of locality, we can determine small independent subsets of events that interfere with each other in a context-dependent way [Missiaen 91b]. Another solution is to add a constraint module that checks whether the abduced temporal relations can be satisfied for a theory of linear order [Denecker Missiaen and B. 92].

Program Transformation The problem of completeness can be avoided by a less natural formulation of a problem domain. For Program 7, termination of property r depends on the presence of property q. Introducing a property **not_q** and writing all *initiates/2* rules explicitly in terms of q and *not_q* guarantees completeness.

```
initiates(start, r).                            succeeds(e1).
initiates(start, q).                            succeeds(e2).
initiates(e1, p) ← holds_at(q, e1).
initiates(e1, p) ← holds_at(not_q, e1).
initiates(e2, not_q).
terminates(e1,r) ← holds_at(q,e1).
terminates(e2,q).

← holds_at(p, t1), holds_at(r, t1).
```

Program 8 Transformation on Program 7

General abductive proof procedure The correctness problem follows from the restricted abductive extension of negation by failure for an event calculus program. The problem of Program 6 follows from the deductive execution of the nested call of *terminates(C,P)* in Rule (2.3). Since this is a positive goal, it should be executed abductively instead of deductively as is done in SLDNFA [Denecker Missiaen and B. 92]. SLDNFA is sound with respect to the completion semantics and it generates at least all minimal solutions if the computation terminates [Denecker and De Schreye 92]. However, the general abductive extension of negation by failure of SLDNFA results in an explosion of the search space. Powerful and intelligent control strategies will be required to make SLDNFA appropriate for solving practical planning problems.

5 Implementation of the abductive proof procedure for event calculus

This section describes the data structures and algorithms of the Prolog implementation of abductive proof procedure for event calculus. The abductive proof procedure is the reasoning component of the prototype planner CHICA. The search component of this planner will be described in Section 6

5.1 Skolem constants

Section 4.2.1 explained the difference between object skolem constants and event skolem constants. The object skolems behave as normal Prolog variables in the unification algorithm extended with the contraint checking mechanism of Δs (Section 4.2.2). On the other hand, event skolems cannot bind to terms, and therefore, they behave as normal Prolog constants during extended unification. The following computation rule contraint is always applicable:

> If a goal contains the literal *happens(***e***)* in which **e** is either a variable or a constant, then *happens(***e***)* must be selected before any other literal *g[***e***]* different from *happens(***e***)*.

This constraint guarantees *happens/1* is the only literal that can be selected with an event variable argument. All the other selected literals will never contain event variables in their arguments. As a result, *happens/1* is the only literal that can introduce new event skolems.

Object skolem constants are represented by Prolog variables. However, the same object skolem in an intermediate goal G^i, in Δp^i, in Δc^i, and in Δs^i, must be represented by the same Prolog variable symbol. Therefore, these components will be grouped into a compound term called *an intermediate solution*: $< G^i, \Delta p^i, \Delta t^i, \Delta c^i, \Delta s^i >$. During a deductive refutation, performed during the persistence operation (Section 4.2.3) and the maintenance operation (Section 4.2.4), skolems must be treated as constants. Therefore, a skolemized copy of the intermediate solution, which replaces every Prolog variable with a unique constant, is used whenever a deductive refutation is performed.

5.2 Implementation of Δs

Δs consists of the inequality constraints and the domain constraints (Section 4.2.2): $\Delta s = \Delta s^{\neq} + \Delta s^{\mathcal{D}}$, $\Delta s^{\neq}$ are the inequality constraints, and $\Delta s^{\mathcal{D}}$ are the domain constraints.

The inequality constraints in $\Delta s^{\neq}$ should be simple inequalities only: $\mathbf{x} \neq \mathbf{y}$, with $\mathbf{x}$, $\mathbf{y}$, or both representing an object skolem constant. For an event calculus program, this can easily be achieved by a normalization procedure which transforms the domain clauses into equivalent clauses that make unifications explicit using *=/2* literals in the body of the clauses [Bruynooghe 86, Kluzniak 87]. Since object skolem constants are represented by Prolog variables, every inequality in $\Delta s^{\neq}$ can be written as $(X \neq \mathbf{y})$ with X a Prolog variable and $\mathbf{y}$ a term distinct from X. $\Delta s^{\neq}$ explicitly represents the symmetric relationship of the inequalities of the form $X \neq Y$, in which X and Y are distinct Prolog variables: both $X \neq Y$ and $Y \neq X$ belong to $\Delta s^{\neq}$.

$\Delta s^{\neq}$ is represented as an association list: $[X_1 - \mathcal{Y}_1, \ldots, X_n - \mathcal{Y}_n]$. A key–value pair, X–$\mathcal{Y}$, has the following meaning.

$$\forall y_i \in \mathcal{Y}, X \neq y_i$$

A domain constraint is of the form $\chi \in \mathcal{D}$ in which χ is a skolem and $\mathcal{D}$ is a set of terms from the Herbrand universe. The meaning of $\chi \in \mathcal{D}$, in which $\mathcal{D} = \{d_1, \ldots, d_n\}$, is,

$$\chi = d_1 \vee \ldots \vee \chi = d_n$$

The skolem constant χ is represented by a Prolog variable. The values of $\mathcal{D}$ are ground terms. $\Delta s^{\mathcal{D}}$ is also represented by an association list of key–value pairs, in which the key is the domain variable and the value is the corresponding domain. The domain is represented as an ordered set. Propagation eliminates redundant representation of inequality constraints on domain variables in both $\Delta s^{\neq}$ and $\Delta s^{\mathcal{D}}$.

During a refutation, the following operations are performed on Δs:

1. The addition of an inequality $\mathbf{x} \neq \mathbf{y}$ to Δs.

2. The addition of a domain constraint $X \in \mathcal{D}$ to Δs.

3. The simplification (and checking) of Δs to take care of the non-monotonicity of Δs with respect to the binding of Prolog variables, which represent object skolems.

All of these operations will either fail or return a new Δs. The mechanism of the operations was explained in Section 4.2.2. Their implementation in terms of the data structures of $\Delta s^{\neq}$ and $\Delta s^{\mathcal{D}}$ uses set operations and has a time complexity of $O(n^2)$, with n the number of elements in $\Delta s^{\neq}$ or $\Delta s^{\mathcal{D}}$[Missiaen 91b].

5.3 Implementation of Δt

A time point in Δt is either a constant term from the Herbrand universe or an event skolem. Event skolems are represented by Prolog constants. Therefore, we represent Δt as a list of the following form: $[t_1 - \mathcal{U}_1, \ldots, t_n - \mathcal{U}_n]$, with $t_1, \ldots, t_n$ distinct time points denoted by Prolog atoms that are lexicographically ordered. This total lexicographical ordering $</2$ should not be confused with the partial precedence ordering on times which is represented by $\prec/2$. Every $\mathcal{U}_i$ is a lexicographicaly ordered set of time points. Every element $t_i - \mathcal{U}_i$ has the following meaning.

$$\forall u_j \in \mathcal{U}_i, t_i \prec u_j$$

The precedence relation $\prec/2$ is transitive, asymmetric and irreflexive. Therefore, Δt must satisfy the following contraints:

(1) if $t_j \in \mathcal{U}_i$ and $t_k \in \mathcal{U}_j$ then $t_k \in \mathcal{U}_i$
(2) if $t_i \in \mathcal{U}_j$ then $t_j \notin \mathcal{U}_i$
(3) $t_i \notin \mathcal{U}_i$

The operations on Δt must keep (1), (2) and (3) satisfied.

During refutation, a selected literal $t_1 \prec t_2$ is either resolved with Δt or it is abduced. Resolution of $t_1 \prec t_2$ can be done in linear time by consulting Δt. Abduction of $t_1 \prec t_2$ can be done by adding $t_1 \prec t_2$ to Δt. The addition algorithm, which is based on Warshall's transitive closure algorithm, is $O(n^2)$ [Missiaen 91b].

The maintenance mechanism of Section 4.2.4 explained the need to remove precedence relations from Δt. However, the removal of a precedence relation from Δt must undo the associated transitive closure relations. This can only be done correctly if a complete history of the additions of precedence relations to Δt is known. As will be explained in the next section, the assumptions made on $\prec/2$ during a persistence or maintenance step are recorded and the transitively closed Δt is reconstructed anew from the accumulated precedence relations that are still valid. This operation costs $O(n^3)$ using Warshall's algorithm. This costly operation is justified because a maintenance step is performed rarely, relative to the number of persistence steps.

5.4 Implementation of Δc

Section 4.2.3 explained the non-monotonicity of *clipped(E,P,T)* with respect to the binding of skolems and the addition of events to Δp. Therefore, the literals of Δc must be rechecked when a variable becomes bound or when an action is added to Δp. This constraint checking is extended with the maintenance mechanism of Section 4.2.4. A maintenance step removes the time relations from Δt that were **only** used, both in resolution and abduction, to construct the failure tree of *not clipped(e,p,t)*. Therefore, alongside Δt, $\Delta t_B \subset \Delta t$ is constructed so that it consists of the abduced time relations in rules different from rule (2.4). In other words, in an intermediate goal, whenever a time relation is refuted that originates from a rule different from rule (2.4) it is resolved with respect to Δt_B. During the finite failure of *clipped(e,p,t)*, a set Δt_k^p is constructed containing the time relations that are used, either in a resolution with Δt or abduced, and do not belong to Δt_B. In other words, Δt_k^p are the assumptions on the precedence relation (different from Δt_B) necessary to construct the finite failure tree of *clipped(e,p,t)*. If successful, id_k–*clipped(e,p,t)* is added to Δc, and id_k–Δt_k^p is added to a separate association list Δt_P. id_k is a unique index associated with every constraint in Δc. In a maintenance step, since id_k–*clipped(e,p,t)* is removed from Δc, the assumptions Δt_k^p must be removed also. However, only the time relations of Δt_k^p can be removed that were **not** used in another resolution step or assumed by another abduction step. Therefore, the transitively closed Δt is reconstructed from the sets of basic time relations Δt_i^p of Δt_P, with $i \neq k$, and from the set Δt_B:

$$\Delta t = transitive\_closure\_of(\Delta t_B + \Delta t_1^p + \ldots + \Delta t_{k-1}^p + \Delta t_{k+1}^p + \ldots + \Delta t_n^p)$$

5.5 Macro refutation step

From the structure of the rules of an event calculus program, it follows that the only recursive temporal predicates are *holds_at/2*, *maintain/3* and *succeeds/1*. A *macro refutation step* will select one of these literals in the goal and perform a number of single refutation steps until a new goal is obtained that consists of temporal recursive literals only. Hence, given a goal that only consists of the predicates *holds_at/2*, *maintain/3* and *succeeds/1*, a macro refutation step will transform this goal into a new goal with the same structure. Δt, Δt_P and Δt_B are put together in one term. For simplicity, we refer to this compound term as the time relations Δt. The predicate *ref_step/10* defines the macro refutation step:

ref_step(G+, Δp*+,* Δt*+,* Δc*+,* Δs*+, Gs-,* Δp_1*-,* Δt_1*-,* Δc_1*,* Δs_1*)*

G is either *holds_at/2*, *maintain/3* or *succeeds/1*. *ref_step/10* performs a number of abductive refutation steps as follows.

$$< G, \Delta p, \Delta t, \Delta c, \Delta s >, \ldots, < G^i, \Delta p^i, \Delta t^i, \Delta c^i, \Delta s^i >, \ldots, < Gs, \Delta p_1, \Delta t_1, \Delta c_1, \Delta s_1 >$$

Gs contains only literals for either *holds_at/2*, *maintain/3* or *succeeds/1*. In every refutation step $i > 1$, a literal is selected different from *holds_at/2*, *maintain/3* and *succeeds/1*. The constraint checking of Δs and Δc and the updating of the components of Δt is performed as explained in the previous sections. *ref_step/10* is a non-deterministic procedure. Because a recursive predicate can be selected only once, the number of solutions generated by *ref_step/10* is finite.

6 Search algorithm

6.1 Proof tree

Given an initial goal G^0, the procedure *search(*G^0*+, Sol-)* is specified as follows:

search(G^0*+, Sol-)*

$Sol =< \Box, \Delta p, \Delta t, \Delta c, \Delta s >$ is a solution node for G^0, i.e.,

- $T + \Delta p + \Delta t \vdash \sigma(G^0)$, for some variable substitution σ
- $T + \Delta p + \Delta t \vdash \Delta c$
- Δs is satisfied

search(G^0, Sol) can be defined using the procedure *tree_search($\mathcal{L}$, Sol)*, which implements a generic search algorithm:

- Initialization:
  Create a root node $< G^0, \Delta p^0, \Delta t^0, \Delta c^0, \Delta s^0 >$, and create a singleton list $\mathcal{L}$ of the root node; call *tree_search($\mathcal{L}$, Sol)*.

- *tree_search($\mathcal{L}$, Sol)*:

  1. if $\mathcal{L}$ is empty then fail
  2. if $\mathcal{L}$ is not empty then
     Select a node $n \in \mathcal{L}$, $\mathcal{L}_1 = \mathcal{L} - n$, $n =< G, \Delta p, \Delta t, \Delta c, \Delta s >$

     a. if $G = \Box$, then $Sol = n$

     b. else if n can be **pruned**, then call *tree_search($\mathcal{L}_1$, Sol)*

     c. else, let $G = g_1, \ldots, g_{k-1}, g_k, g_{k+1}, \ldots, g_n$,
        - **select** a literal $g_k \in G$
        - Perform all macro refutation steps starting from g_k:
          findall($< (g_1, \ldots, g_{k-1}, Ls, g_{k+1}, \ldots, g_n), \Delta p_1, \Delta t_1, \Delta c_1, \Delta s_1 >$,
          ref_step(g_k, Δp, Δt, Δc, Δs, Ls, Δp_1, Δt_1, Δc_1, Δs_1),
          $\mathcal{S}$)
        - $\mathcal{L}_2 = \mathcal{L}_1 + \mathcal{S}$
        - call *tree_search($\mathcal{L}_2$, Sol)*

The nodes in the proof tree correspond to plans and the branches correspond to macro refutation steps. Finding a successful macro refutation corresponds to finding *a solution node*, a node with an empty goal. The search algorithm will calculate a part of the search tree explicitly until a solution node is found. After each macro refutation step, the search must decide which leaf node to continue from. This is called the *node selection.* A node can be *pruned* from the search tree based on pruning heuristics; this node and all of its successor nodes are then no longer considered. At a selected leaf node that is not pruned, the search algorithm must select a goal literal, which corresponds to the *computation rule.*

6.2 Search strategy

The proof tree of Section 6.1 resembles the search space of a refutation proof in logic programming. The procedure *tree_search($\mathcal{L}$, Sol)* implements a generic search algorithm. Depending on the particular choice of the node selection, the computation, and the pruning rules, *tree_search($\mathcal{L}$, Sol)* can implement any uninformed search strategy and any heuristic search procedure.

6.2.1 Computation rule

The goal literals that can be selected are *succeeds/1*, *maintain/3* and *holds_at/2*. The default selection is left-to-right, but first the literals *succeeds/1*, then the literals *maintain/3*, and finally the literals *holds_at/2*. The left-to-right selection of the goals *holds_at(P,T)* can be overruled by specifying a priority number N for each property P in the domain definition of the problem. For example, in the robot block world of Program 3 we add the following definitions:

{*a low number indicates a high priority*}
prop_priority(on(X,Y), 1).
prop_priority(clasped(X), 1).
prop_priority(clear(X), 2).
prop_priority(free_robot, 3).

6.2.2 Node selection rule

The default selection rule is depth-first; this can be overruled by calculating a priority number for the parial plans Δp in the search space: *priority_of_plan*(Δp+, *N-*). The predicate *priority_of_plan/2* is defined in the problem domain definition. This simple scheme is adequate in most cases, but any other domain specific node selection strategy can be implemented.

6.2.3 Pruning a node

Pruning heuristics reduce the search space. The domain dependent heuristics must detect plans Δp that cannot lead to a valid plan or that will lead to an undesirable plan such as a non-optimal plan. For example, in the robot block world, a plan in which the robot picks the same block more than twice is not optimal:

prune_plan(Plan) $\leftarrow$ block(X), number_of_events_of_act(Plan, pick(X), P), P > 2.

7 CHICA compared with other planning systems

CHICA is the first practical planner using the abductive event calculus. ABPLAN can be considered a theoretical predecessor of CHICA [Esghi 88]. ABPLAN's event calculus uses a STRIPS-like representation for the pre- and postconditions of an action, and as a result, cannot express context dependent events. ABPLAN's reasoning component is based on constraint solving which avoids backtracking by removing previously generated assumptions. Esghi suggests using an assumption based truth maintenance system for this purpose, but such an implementation is unknown to the authors. It is not clear how ABPLAN obtains completeness by controlling the search for a solution. Shanahan proposed improvements to ABPLAN's event calculus by adding a persistence axioms, thus eliminating ABPLAN's meta-level integrity constraints [Shanahan 89]. We further improved the event calculus for planning by adding a predicate *succeeds/1* for expressing an action's preconditions and by adding a maintenance axiom in order to reduce backtracking. Unlike ABPLAN, CHICA's reasoning component is based on a sound abductive proof procedure, SLDNFA. CHICA selects non-ground literals and binds skolems using extended unification whereas ABPLAN abduces equality assumptions explicitly. CHICA is the first planner to present the correctness and completeness problems of context-dependent actions and derived properties in planning within a logic programming framework.

Most domain-independent planners descend from STRIPS [Fikes and Nilsson 71]. In STRIPS, an action is associated with an add-and-delete list of properties. These properties are added to or deleted from the world state in which the action is applied. Plan modification is obtained by inserting a new action, imposing an ordering constraint, or instantiating a variable. Event calculus in Horn clause logic is more expressive than a STRIPS representation or any of its extensions. For example, none of the STRIPS' extensions can represent and derive truth values of derived properties. All properties must be represented explicitly in the add and delete lists of the actions, which complicates validation and maintenance of the domain description of real world problems.

CHICA's plan modification is achieved by the abductive proof procedure. Inserting a new action and imposing an ordering constraint correspond to abducing facts in Δ. In CHICA, instantiating a variable is performed by extended unification, whereas in other planners this is done by a complex, often ill-defined, matching algorithm. Constraint propagation as in MOLGEN [Stefik 81] and SIPE [Wilkins 88], is done in CHICA by means of finite domain constraint logic programming techniques.

CHICA's goal directed search mechanism follows from the abductive proof procedure. For practical planning problems, CHICA's search can be controlled by heuristics defining computation, selection, and pruning rules. Comparison of CHICA's search mechanism with other AI planners is difficult because search control is the least documented aspect of AI planners: TWEAK [Chapman 87] and Prodigy [Fink and Veloso 94] are exceptions. TWEAK is a theoretical planning implementation that uses dependency-directed breadth-first search. Prodigy's planning algorithm uses a combination of backward chaining non-linear planning and forward chaining linear planning. Prodigy could accommodate for heuristic rules to decide which branch of the search space to explore next. In addition, Prodigy uses learning procedures to improve the efficiency of the planner. In a similar way, CHICA has been used as the planning component of an intelligent autonomous agent [Sablon and Bruynooghe 94].

8 Example planning problems

The prototype planner CHICA has been applied to three problem domains: the multiple robot block world, the assembly of a flashlight, and a room decoration problem[5].

8.1 Multiple robot block world

The multiple robot block world is an extension of the robot block world of Program 3. The robot actions *pick* and *put* have an extra parameter designating the robot performing the action, for example *pick(X,R)* names the action that the robot R picks the block X. *robot/1*

[5]CHICA was implemented in ProLog by BIM under SunOS 4.2; it has an OpenLook graphical user interface.

is the unary predicate that defines the actual robots in the given problem. The property *free_robot* is replaced by *free(R)*, designating that the robot R is free.

CHICA solves different three and four block problems with one, two and three robots. Two simple pruning heuristics are used: a robot should not pick up the same block more than twice and a robot may not put a block on the table more than once. The predicate *priority_of_plan/2* prioritizes the solution that uses the largest number of robots in its plan. The computation rule is determined by the definitions of *prop_priority/2*.

8.2 Multiple robot assembly problem

The flashlight planning problem is an example of an assembly problem [Sanderson et al. 90, Missiaen 91a]. The flashlight assembly has four parts: *cap, stick, receptacle,* and *handle.* The two ends of the receptacle are designated as *top(receptacle)* and *bottom(receptacle).* The cap must be screwed onto the top of the receptacle, the handle must be screwed onto the bottom of the receptacle, and the stick must inserted into the receptacle; these goal conditions are represented respectively as *screw_on(cap, top(receptacle)), screw_on(handle, bottom(receptacle))* and *in(stick, receptacle).* Initially, all parts are disassembled. The bottom of the receptacle must be closed by the handle before the stick can be put into the receptacle.

The robot cell consists of a number of robots which share a common feeder. A robot can perform the following actions:

- insert a part into another part that is held by another robot;
- screw a part onto the top or onto the bottom of another part held by another robot;
- pick up a part that is provided by the feeder;
- release a part onto the table;
- pick up a part from the table.

The feeder is a stack device that gives one part at a time to the robot.

Pruning rules eliminate infinite branches in the search space, for example:

{*Robot can get the same part from the feeder only once*}
prune_plan(R, Plan) ←
    robot(R), part(O), number_of_events_of_act(Plan, get(O, R), N), N > 1.

No definitions for *prop_priority/2* and *priority_of_plan/3* are given which result in a depth-first, left-to-right execution. The node selection and the computation rule follow from the ordering of the domain rules and from the ordering of the goals in the body of these rules. For example, if a robot tries to get hold of a part, it first tries to get that part from the feeder before it tries to pick that part from the table. Resolution steps are always performed before abduction steps; as a result, existing actions are used before new actions to satisfy a goal.

CHICA solves the flashlight assembly problem for a robot cell with two robots in 64 seconds on a Sun SPARCstation 1+; 53 nodes have been created in the search tree; persistence constraints in Δc were checked 536 times, of which 28 resulted in the application of a maintenance step thus avoiding backtracking.

8.3 Room finishing problem

The room finishing problem is the largest problem that has been solved so far. The finishing involves papering the walls, making electrical connections, putting water pipes in the walls and so on. The room is empty except for one or more tables that can be moved by the workers. The tables are positioned against the walls, and can be moved from one wall to another by a worker; at most one table can be placed against a particular wall. A worker is a plumber, a paper hanger, or an electrician. A plumber connects a water pipe to the water supply after the water pipe has been inserted into a wall by any of the workers. Therefore, a worker drills a hole in the wall and inserts the pipe; the drilling of a hole makes the wall dirty. A plumber can also supply and cut off the water to the room. A paper hanger glues a sheet of paper using the table and hangs it on the wall after the wall has been cleaned by any of the workers; no table may be in front of a wall being papered. An electrician connects two walls electrically using wiring pipes.

The problem domain is described by 16 properties, 12 types of actions, 19 *initiates* rules, 16 *terminates* rules and 12 *succeeds* rules; for example:

succeeds(E) ←
  act(E,insert_water_pipe(W,PL,WP)), wall(W), plumber(PL), water_pipe(WP),
  holds_at(new_pipe(WP), E), holds_at(hole_in(WP,W), E), holds_at(available(W,PL), E).

Ten different pruning rules are defined, and 16 *prop_priority/2* facts determine the computation rule. CHICA finds a solution with 24 events in 318 seconds. This long execution time results from the large search space that was generated: 4565 nodes. Powerful heuristics are the only way to reduce the search space. Drawing up and testing heuristics for real world planning problems is not trivial: it took Wilkins several months to implement a production line planner for a large Australian brewery [Wilkins 90].

9 Extensions and Future Research

9.1 Plan execution and replanning

CHICA is a preplanner that calculates a plan prior to its execution. During execution, all planned events should be executed and no events external to the plan should happen, or else the plan might not achieve its planning goal. In the real world, the execution component could encounter unplanned events and execution of the planned events could be unsuccessful. These cases must be detected by an execution monitor equiped with sensory devices, and the effect on the plan must be computed. If the plan is no longer valid, it should change or be recalculated. In an emergency, a reactive planning component would take appropriate action immediately.

Such extensions to preplanning can be solved within the same logical framework [Esghi 88, Missiaen 91b]. If the execution component executed the plan $\Delta p + \Delta t$ associated with a set Δc, then the elements of Δc would be constraints of the form *not clipped(e,p,t)*. During execution, such a constraint would be active if the current time were within the interval [e,t]. Sensing the property p to be false would indicate that an external event had occurred, or a planned event had failed. In this way, a possible plan failure could be detected as soon as possible. The maintenance mechanism of Section 4.2.4 could change the plan to restore the violated constraint before continuing execution. If this mechanism fails, a new plan would need to be generated for the remaining goals.

9.2 Compound Actions and Intervals

In event calculus, actions are assumed to occur instantaneously. However, it would be more realistic to associate a duration or an interval with an action and to allow every action to decompose into more elementary actions. We could model an interval in event calculus with its start and end points. However, Allen advocates the use of an explicit representation of compound actions and intervals [Allen 83, Allen 84]. Allen's theory can be represented in logic [Sadri 87], and it would be interesting to apply the abductive proof procedure to this theory and use it for plan generation. This would result in a planning system with a more expressive representation language.

10 Conclusion

This article describes Artificial Intelligence planning within a logic programming framework.

We have presented a logical theory based on event calculus for representing planning problems and reasoning about time and change. Plan generation in event calculus is obtained through the application of the abductive proof procedure, an extension of SLDNF resolution, specialized for event calculus. This procedure generates assumptions in order to prove a given goal. A constraint mechanism deals with the non-monotonicity of a property's persistence over an interval. A plan modification mechanism is used to reduce backtracking. The use of domain constraints and CLP techniques for checking these constraints improves the efficient treatment of inequality constraints. The prototype implementation CHICA demonstrates the possibility of building a practical AI-planner using techniques of computational logic, including resolution, extended unification, abduction and constraint logic programming. Powerful domain heuristics are required to solve non-trivial planning problems; these heuristics prune the search space, and determine the node selection and the computation rule of CHICA's generic search algorithm. CHICA has been applied successfully to solve planning problems in three non-trivial planning domains.

Plan execution, replanning and reactive planning are possible extensions to closed-world preplanning that can be explored within the same logical framework of this article.

Acknowledgments

Fariba Sadri improved the quality of this manuscript indirectly by her many useful comments on the doctoral work preceding this manuscript. We thank Gunther Sablon for correcting the first draft of this manuscript, and Robert Main and Jon Wilkes for correcting the final version of this manuscript. Franciska Soete nicknamed the planner implementation, CHICA.

References

[Allen 83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[Allen 84] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[Borillo and Gaume 90] Mario Borillo and Bruno Gaume. An extension to Kowalski and Sergot's event calculus. In *Proceedings ECAI 90*, pages 99–104, August 1990.

[Bruynooghe 86] Maurice Bruynooghe. Compile time garbage collection. Report CW43, Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium, 1986.

[Bry Manthey and M. 90] François Bry, Rainer Manthey, and Bern Martens. Integrity verification in knowledge bases. Technical Report D.2.1.a, ECRC, April 1990.

[Chapman 87] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[Clark 78] Keith. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[Cox and Pietrzykowski 86] P. T. Cox and T. Pietrzykowski. Causes for events: their computation and applications. In J.H. Siekmann, editor, *Proceedings CADE-86*, Lecture Notes in Computer Science, pages 608–621. Springer-Verlag, 1986.

[Denecker and De Schreye 92] Marc Denecker and Danny De Schreye. SLDNFA; an abductive procedure for normal abductive programs. In K. Apt, editor, *Proceedings International Joint Conference and Symposium on Logic Programming, Washington*, 1992.

[Denecker Missiaen and B. 92] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proceedings of the European Conference on Artificial Intelligence (ECAI92), Vienna*, pages 384–388, August 1992.

[Esghi 88] Kave Esghi. Abductive planning with the event calculus. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming 1: 5th ICLP*, pages 562–579, 1988.

[Esghi and Kowalski 89] Kave Esghi and Robert A. Kowalski. Abduction compared with negation by failure. In *Proceedings of the 6th International Conference on Logic Programming*, pages 234–255. The MIT Press, 1989.

[Fikes and Nilsson 71] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[Finger and Genesereth 85] J.J. Finger and M.R. Genesereth. Residue: A deductive approach to design synthesis. Technical Report STAN-CS-85-1035, Stanford University, Stanford, California, 1985.

[Fink and Veloso 94] Eugene Fink and Manuela Veloso. Prodigy planning algorithm. Technical Note CMU-CS-94-123, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1994.

[Kakas and Mancarella 90] A. C. Kakas and P. Mancarella. Database updates through abduction. In *Proceedings VLDB 90*, pages 650–661, August 1990.

[Kluzniak 87] Feliks Kluzniak. Type synthesis for ground prolog. In J.-L. Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming, Melbourne, 1987*, pages 788–816. The MIT Press, 1987.

[Kowalski and Sergot 86] Robert A. Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[McCarthy and Hayes 69] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[Missiaen 89] Lode R. Missiaen. Situation calculus and event calculus in modal logic Z. Report CW85, Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium, March 1989.

[Missiaen 91a] Lode R. Missiaen. Localized abductive planning for robot assembly. In *Proceedings 1991 IEEE Conference on Robotics and Automation*, pages 605–610. IEEE Robotics and Automation Society, April 1991.

[Missiaen 91b] Lode R. Missiaen. *Localized Abductive Planning with the Event Calculus*. PhD thesis, KU Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Leuven, Belgium, September 1991.

[Sablon and Bruynooghe 94] Gunther Sablon and Maurice Bruynooghe. Using the event calculus to integrate planning and learning in an intelligent autonomous agent. In C. Bäckström and E. Sandewall, editors, *Current Trends in AI Planning*, pages 254–265. IOS Press, 1994.

[Sadri 87] Fariba Sadri. Three recent approaches to temporal reasoning. In Antony Galton, editor, *Temporal Logics and their Applications*, pages 121–168. Academic Press, 1987.

[Sadri and Kowalski 88] Fariba Sadri and Robert A. Kowalski. A theorem proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 313–362. Morgan Kaufmann Publishers, Inc., 1988.

[Sanderson et al. 90] Arthur C. Sanderson, Luiz S. Homem de Mello, and Hui Zhang. Assembly sequence planning. *AI Magazine*, 11(1):62–81, 1990.

[Shanahan 89] Murray P. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings IJCAI 89*, page 1055. International Joint Conference on Artificial Intelligence, 1989.

[Shanahan 90] Murray P. Shanahan. Representating continuous change in the event calculus. In *Proceedings ECAI 90*, pages 598–603, August 1990.

[Sripada 91] Sury M. Sripada. *Temporal Reasoning in Deductive Databases*. PhD thesis, Imperial College, Department of Computing, 180 Queen's Gate, London SW7 2BZ, England, 1991.

[Stefik 81] Mark Jeffrey Stefik. Planning with constraints. *Artificial Intelligence*, 16, 1981.

[Sussman 73] Gerald Jay Sussman. A computational model of skill acquisition. Technical Report MIT AI 279, MIT, August 1973.

[Van Hentenryck 89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[Wilkins 88] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers, Inc., 1988.

[Wilkins 90] David E. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 4(4):232–246, November 1990.