Abductive Logic Programming with CIFF: Implementation and Applications

U. Endriss[1], P. Mancarella[2], F. Sadri[1], G. Terrreni[2], and F. Toni[1,2]

[1] Department of Computing, Imperial College London
Email: {ue,fs,ft}@doc.ic.ac.uk
[2] Dipartimento di Informatica, Università di Pisa
Email: {paolo,terreni,toni}@di.unipi.it

Abstract. We describe a system implementing a novel extension of Fung and Kowalski's IFF abductive proof procedure which we call CIFF, and its application to realise intelligent agents that can construct (partial or complete) plans and react to changes in the environment. CIFF extends the original IFF procedure in two ways: by dealing with constraint predicates and by dealing with non-allowed abductive logic programs.

1 Introduction

Abduction has long been recognised as a powerful mechanism for hypothetical reasoning in the presence of incomplete knowledge [9, 13]. In this paper, we discuss the implementation and applications of a novel abductive proof procedure, which we call CIFF. This procedure extends the IFF proof procedure of Fung and Kowalski [10] and is described in detail in [8].

A number of abductive proof procedures have been proposed in the literature [12]. Kakas and Mancarella [13] extended the (first ever) abductive proof procedure of Eshghi and Kowalski [9] for normal logic programming to Abductive Logic Programming (ALP). This has been augmented to deal with constraint predicates in [15] and with integrity constraints that behave like condition-action rules in [22], and has been reformulated to improve its complexity in [21]. All these procedures are proved correct wrt. the (partial) stable model semantics. Another "family" of abductive proof procedures extend that of Console et al. [5] and are proved correct wrt. the completion semantics [4, 20]: these are the IFF procedure [10], and its extensions to deal with constraint predicates, treated as abducibles [19], and integrity constraints which behave like condition-action rules [24]; and the SLDNFA procedure [6], and its extensions to deal with constraint predicates [7]. Some of these procedures have been implemented and experimented with in a number of applications, e.g. [15]. Moreover, recently a system combining features of ACLP [15] and SLDNFA [6], and using heuristic search to improve efficiency, has been proposed in the form of the A-system [17].

Our interest in combinations of ALP and constraint reasoning stems from our work on using computational logic-based techniques in the area of multiagent systems and global computing (for instance, to implement an agent's planning

capability) [14, 27]. We have found, however, that our requirements for these applications go beyond available state-of-the-art ALP proof procedures. This consideration has led us to devise the CIFF proof procedure [8]. The novelty of CIFF is twofold: (1) it dynamically deals with non-allowed programs (i.e. programs with problematic quantification patterns that cannot be handled by the original IFF procedure), thus having wider applicability; and (2) it extends IFF by integrating the procedure with a "black box" constraint solver (rather than treating constraint predicates as abducibles as in [19]). Moreover, CIFF inherits from the original IFF procedure its forward reasoning as well as its syntax of integrity constraints (which is more general than that of most other procedures) and its flexible handling of variables, all of which have been listed as advantages of IFF over other proof procedures. In the present paper, we describe the first implementation of CIFF and illustrate one of its potential applications, namely planning, in some detail. Differently from conventional practice in logic programming, we consider *partial* (rather than *complete*) planning as well as plan repair (via reactivity) in a dynamic environment. We have used CIFF in the PROSOCS system [27], a computational logic-based platform for programming intelligent agents, to build the planning and reactivity components of an agent.

The remainder of this paper is structured as follows. Section 2 provides a brief introduction to ALP and reviews the definition of the CIFF proof procedure. This theoretical presentation is complemented by Section 3, where we show how we have implemented the procedure in Prolog. The application of CIFF to planning is discussed in Section 4. Section 5 concludes.

2 Abductive Logic Programming with CIFF

In this section, we briefly review the framework of Abductive Logic Programming (ALP) for knowledge representation and reasoning [12], as well as the CIFF proof procedure for ALP [8]. ALP can be usefully extended with the handling of constraint predicates in the same way as Constraint Logic Programming (CLP) [11] extends logic programming (see e.g. [15, 19]). Throughout this paper, we assume that our language includes a number of constraint predicates.

2.1 Abductive Logic Programming with Constraints

An *abductive logic program* is a triple $\langle P, I, A\rangle$, where P is a normal logic program (with constraints), I is a finite set of sentences in the language of P (called *integrity constraints*), and A is a set of abducible predicates in the language of P, not occurring in the head of any clause of P [12]. A *query* Q is a conjunction of literals. Any variables occurring in Q are implicitly existentially quantified. These variables are also called the *free* variables.

Broadly speaking, given a program $\langle P, I, A\rangle$ with constraint predicates and a query Q, the purpose of abduction is to find a (possibly minimal) set of abducible atoms (namely atoms whose predicate is abducible) Δ which, together with P and the constraint structure over which the constraint predicates are defined [11],

"entail" (an appropriate ground instantiation of) Q, with respect to a suitable notion of "entailment", in such a way that the extension of P with this set "satisfies" I (see [12] for possible notions of integrity constraint satisfaction). The appropriate notion of "entailment" depends on the semantics associated with the logic program P (again, there are several possible choices for such a semantics [12]). In the remainder of this paper we will adopt the *completion semantics* [4, 20] for logic programming, and extend it *à la* CLP to take the constraint structure into account. We represent entailment under such semantics as $\models_{\Re}$. An *abductive answer* to a query Q for a program $\langle P, I, A\rangle$, containing constraint predicates defined over a structure $\Re$, is a pair $\langle \Delta, \sigma\rangle$, where Δ is a set of ground abducible atoms and σ is a substitution for the free variables in Q such that $P \cup \Delta\sigma \models_{\Re} I \wedge Q\sigma$.

2.2 The CIFF Proof Procedure

We now give a brief description of the CIFF procedure [8]. Like Fung and Kowalski [10], we require the theory given as input to be represented in "iff-form" [4, 10], which we can obtain by *selectively completing* P with respect to all predicates except *special* predicates (*true*, *false*, constraint and abducible predicates). That is, an alternative representation of an abductive logic program would be a pair $\langle Th, I\rangle$, where Th is a set of (homogenised) iff-definitions:

$$p(X_1, \ldots, X_k) \Leftrightarrow D_1 \vee \cdots \vee D_n$$

Here, p must not be a special predicate and there can be at most one iff-definition for every predicate symbol. Each of the disjuncts D_i is a conjunction of literals. The variables $X_1, \ldots, X_k$ are implicitly universally quantified with the scope being the entire definition. Any other variable is implicitly existentially quantified, with the scope being the disjunct in which it occurs.

In this paper, the set of integrity constraints I are implications of the form:

$$L_1 \wedge \cdots \wedge L_m \Rightarrow A_1 \vee \cdots \vee A_n$$

Each of the L_i must be a literal; each of the A_i must be an atom. Any variables are implicitly universally quantified with the scope being the entire implication.

In CIFF, the search for abductive answers of queries over a proof tree is initialised with the root of the tree consisting of the integrity constraints in the program and the literals of the query. The procedure then repeatedly manipulates the current node by applying equivalence-preserving *proof rules* to it. The nodes are sets of formulas (the so-called *goals*) which may be atoms, implications, or disjunctions of literals. The implications are either integrity constraints, their residues, or obtained by rewriting negative literals (e.g. *not* p is rewritten as $p \Rightarrow$ *false*.) The proof rules modify nodes by rewriting goals in them, adding new goals to them, or deleting superfluous goals from them. The set of iff-definitions is kept in the background and is only used to *unfold* defined predicates.

Fung and Kowalski [10] require inputs to their proof procedure to meet a number of allowedness conditions (essentially avoiding certain problematic

patterns of quantification) to be able to guarantee its correct operation. These conditions are overly restrictive; IFF could produce correct answers also for some non-allowed inputs. Unfortunately, it is difficult to formulate appropriate allowedness conditions that guarantee a correct execution of the proof procedure without imposing too many unnecessary restrictions. Our proposal, put forward in [8], is to tackle the issue of allowedness *dynamically*, i.e. at runtime, rather than adopting a static and overly strict set of conditions. To this end, CIFF includes a *dynamic allowedness rule* amongst its proof rules, which gets triggered once the procedure encounters, in the current node, formulas it cannot manipulate correctly due to a problematic quantification pattern. When this happens, the node is labelled as *undefined* and will not be selected again. In addition to the dynamic allowedness rule, the main proof rules of CIFF are:

- *Unfolding:* Replace any atomic goal $p(\vec{t})$, for which there is a definition $p(\vec{X}) \Leftrightarrow D_1 \vee \cdots \vee D_n$ in *Th*, by $(D_1 \vee \cdots \vee D_n)[\vec{X}/\vec{t}]$. There is a similar rule for defined predicates inside implications.
- *Splitting:* Rewrite any node with a disjunctive goal as a disjunction of nodes.
- *Propagation:* Given goals of the form $p(\vec{t}) \wedge A \Rightarrow B$ and $p(\vec{s})$, add the goal $(\vec{t} = \vec{s}) \wedge A \Rightarrow B$.
- *Case analysis for constraints:* Replace any goal of the form $Con \wedge A \Rightarrow B$, where *Con* is a constraint not containing any universally quantified variables, by $[Con \wedge (A \Rightarrow B)] \vee \overline{Con}$. There is a similar case analysis rule for equalities.
- *Constraint solving:* Replace any node containing an unsatisfiable set of constraints (as atoms) by *false*.

In addition, there are logical simplification rules, rules for rewriting equalities and making substitutions, and rules that reflect the interplay between constraint predicates and the usual equality predicate. For full details we refer to [8].

In a proof tree for a query, a node containing *false* is called a *failure node*. If all leaf nodes are failure nodes, then the search is said to *fail*. A node to which no more proof rules can be applied is called a *final node*. A final node that is not a failure node and which has not been labelled as *undefined* is called a *success node*. If a success node exists, then the search is said to *succeed*. CIFF has been proved *sound* in [8]: it is possible to extract an abductive answer from any success node (*soundness of success*); and if the search fails then there exists no such answer (*soundness of failure*).

3 Implementation of the Proof Procedure

We have implemented the CIFF procedure in Sicstus Prolog [28].[3] Most of the code could very easily be ported to any other Prolog system conforming to standard Edinburgh syntax. A minor exception is the module concerned with constraint solving as it relies on Sicstus' built-in constraint logic programming solver over finite domains (CLPFD) [3]. However, the modularity of our implementation would make it relatively easy to integrate a different constraint solver

[3] The system is available at `http://www.doc.ic.ac.uk/~ue/ciff/`

```
lamp(X) iff [[X=a]].
battery(X,Y) iff [[X=b, Y=c]].
faulty(X) iff [[lamp(X), broken(X)], [power_failure(X), not(backup(X))]].
backup(X) iff [[battery(X,Y), not(empty(Y))]].
```

Table 1. The abductive logic program for the faulty-lamp example of [10]

into the system instead. The only changes required would be an appropriate re-implementation of a handful of simple predicates providing a wrapper around the constraint solver chosen for the current implementation.

This section discusses various aspects of our implementation of the CIFF procedure and explains how to use the system in practice.

3.1 Representation of Abductive Logic Programs

The CIFF procedure is defined over (selectively) completed logic programs, i.e. sets of definitions in iff-form rather than rules (in if-form) and facts. As these definitions can become rather long and difficult to read, our implementation includes a simple module that translates logic programs into completed logic programs which are then used as input to the CIFF procedure. Being able to complete logic programs on the fly also allows us to spread the definition of a particular predicate over different knowledge bases.

The syntax chosen to represent facts and rules of a logic program is that of Prolog, except that negative literals are represented as Prolog terms of the form `not(`A`)`. In addition, we also allow for (arithmetic) constraints as subgoals in the condition of a rule. For the current implementation, admissible constraints are terms such as `T1 #< T2 + 5`. The available constraint predicates are `#=`, `#\=`, `#<`, `#=<`, `#>`, and `#>=`, each of which takes two arguments that may be any arithmetic expressions over variables and integers (using operators such as addition, subtraction, and multiplication, or any other arithmetic operation that the CLPFD module of Sicstus Prolog can handle [3]). Note that for equalities over terms that are not arithmetic terms, the usual equality predicate `=` should be used (e.g. `X = bob`). Iff-definitions are terms of the form A `iff` B, where A is an atom and B is a list of lists of literals (representing a disjunction of conjunctions). Integrity constraints are expressions of the form A `implies` B, where A is a list of literals (representing a conjunction) and B is a list of atoms (representing a disjunction). Table 1 shows an example using our syntax.

3.2 Running the Program

The main predicate of our implementation is called `ciff/4`:

```
ciff( +Defs, +ICs, +Query, -Answer).
```

The first argument is a list of iff-definitions, the second is a list of integrity constraints, and the third is the list of literals in the query. The `Answer` consists of three parts: a list of abducible atoms, a list of restrictions on the answer

substitution, and a list of constraints (the latter two can be used to construct an answer substitution according to the semantics of ALP).

Alternatively, the first two arguments of `ciff/4` may be replaced with the name of a file containing an abductive logic program. An example for such a file is given in Table 1. This is the *faulty-lamp* example discussed, amongst others, by Fung and Kowalski [10]. The syntax is almost self-explanatory (recall that a list of lists represents a disjunction of conjunctions). This program happens to consist only of iff-definitions (there are no integrity constraints). Assuming that the program is stored in a file called `lamp.alp`, we may run the following query:

```
?- ciff( 'lamp.alp', [faulty(X)], Answer).
Answer = [broken(a)]:[X/a]:[] ? ;
Answer = [empty(c),power_failure(b)]:[X/b]:[] ? ;
Answer = [power_failure(X)]:[not(X/b)]:[] ? ; No
```

Here the user has enforced backtracking, so all three answer are being reported by the system. Note that the third (empty) list in each of the answers would be used to store the associated constraints (of which there are none in this example). For details on how to interpret the above answers, we refer to [10].

3.3 Implementation of the Proof Rules

We are now going to turn our attention to the actual implementation of the proof procedure and explain some of the design decisions taken during its development. Our implementation of CIFF has been inspired by work in the Automated Reasoning community on so-called *lean* theorem provers [1]. Our proof procedure manipulates a list of formulas rather than submitting these formulas *themselves* to the Prolog interpreter. One advantage of this approach is, for instance, that we can *report* variable substitutions at the meta-level rather than having Prolog making the actual instantiations (which would be problematic as CIFF computes only restrictions on the answer substitution, rather than an actual substitution).

The proof rules are repeatedly applied to the current node. Whenever a disjunction is encountered, it is split into a set of successor nodes (one for each disjunct). The procedure then picks one of these successor nodes to continue the proof search and backtracking over this choicepoint results in all possible successor nodes being explored. In theory, the choice of which successor node to explore next is taken nondeterministically; in practice we simply move through nodes from left to right. The procedure terminates when no more proof rules apply (to the current node) and finishes by extracting an answer from this node. Enforced backtracking will result in the next branch (if any) of the proof tree being explored, i.e. in any remaining abductive answers being enumerated. The Prolog predicate implementing the proof rules has the following form:

```
sat( +Node, +EV, +CL, +LM, +Defs, +FreeVars, -Answer).
```

`Node` is a list of goals, representing a conjunction. `EV` is used to keep track of existentially quantified variables in the node. This set is relevant to assess the applicability of some of the proof rules. `CL` (for constraint list) is used to store the

constraints that have been accumulated so far. The next argument, `LM` (for loop management), is a list of expressions of the form $A:B$ recording pairs of formulas that have already been used with particular proof rules, thereby allowing us to avoid loops that would result if these rules were applied over and over to the same arguments (this is necessary, for instance, for the *propagation* rule). This information can also be exploited to improve efficiency by identifying redundant proof steps. `Defs` is the list of iff-definitions in the theory. `FreeVars` is used to store the list of free variables. Finally, running `sat/7` will result in the variable `Answer` to be instantiated with a representation of the abductive answer found by the procedure.

Each proof rule corresponds to a Prolog clause in the implementation of `sat/7`. For example, the *unfolding rule for atoms* is implemented as follows:

```
sat( Node, EV, CL, LM, Defs, FreeVars, Answer) :-
  member( A, Node), is_atom( A), get_def( A, Defs, Ds),
  delete( Node, A, Node1), NewNode = [Ds|Node1], !,
  sat( NewNode, EV, CL, LM, Defs, FreeVars, Answer).
```

The auxiliary predicate `is_atom/1` will succeed whenever the argument represents an atomic goal. Furthermore, `get_def(A,Defs,Ds)`, with the first two arguments being instantiated at the time of calling the predicate, will instantiate `Ds` with the list of lists representing the disjunction that defines the atom `A` according to the iff-definitions in `Defs` whenever there *is* such a definition (i.e. the predicate will fail for abducibles). Once `get_def(A,Defs,Ds)` succeeds we definitely know that the unfolding rule is applicable: there exists an atomic conjunct `A` in the current `Node` and it is not abducible. The cut in the penultimate line is required, because we do not want to allow any backtracking over the *order* in which rules are being applied. After we are certain that this rule should be applied we manipulate the current `Node` and generate its successor `NewNode`. We first delete the atom `A` and then replace it with the disjunction `Ds`. The predicate `sat/7` then recursively calls itself with the new node.

3.4 Testing

The Prolog clauses in the implementation of `sat/7` may be reordered almost arbitrarily (the only requirement is that the clause used to implement answer extraction is listed last). Each order of clauses corresponds to a different proof strategy, as it implicitly assigns different priorities to the different proof rules. This feature of our implementation allows for an experimental study of which strategies yield the fastest derivations. We hope to be able to exploit this feature of the implementation in our future work to study possible optimisation techniques. The order in which proof rules are applied in the current implementation follows some simple heuristics. For instance, logical simplification rules as well as rules to rewrite equality atoms are always applied first. Splitting, on the other hand, is one of the last rules to be applied.

The implementation of the CIFF proof procedure has been tested successfully on a number of examples. Most of these examples are taken from applications

$holds(F, T_2) \leftarrow happens(A, T_1)$, $initiates(A, T_1, F)$, $not\ clipped(T_1, F, T_2)$, $T_1 < T_2$
$holds(F, T) \leftarrow initially(F)$, $not\ clipped(0, F, T)$, $0 < T$
$holds(\neg F, T_2) \leftarrow happens(A, T_1)$, $terminates(A, T_1, F)$, $not\ declipped(T_1, F, T_2)$, $T_1 < T_2$
$holds(\neg F, T) \leftarrow initially(\neg F)$, $not\ declipped(0, F, T)$, $0 < T$
$clipped(T_1, F, T_2) \leftarrow happens(A, T)$, $terminates(A, T, F)$, $T_1 \leq T < T_2$
$declipped(T_1, F, T_2) \leftarrow happens(A, T)$, $initiates(A, T, F)$, $T_1 \leq T < T_2$
$happens(A, T) \leftarrow assume\_happens(A, T)$

Table 2. Domain-independent rules in P_{plan}

$holds(F, T)$, $holds(\neg F, T) \Rightarrow false$
$assume\_happens(A, T)$, $precondition(A, P) \Rightarrow holds(P, T)$
$assume\_happens(A, T_2)$, $not\ executed(A, T_2)$, $time\_now(T_1) \Rightarrow T_1 < T_2$

Table 3. Domain-independent integrity constraints in I_{plan} (for complete planning)

of CIFF within the SOCS project, which investigates the application of computational logic-based techniques to multiagent systems (e.g. the implementation of an agent's planning and a reactivity capabilities). While these are encouraging results, it should also be noted that this is only a first prototype and more research into proof strategies for CIFF as well as a fine-tuning of the implementation are required to achieve satisfactory runtimes for larger examples.

4 An Application to Abductive Planning

In this section, as an example application of the CIFF system, we consider how it can be used for planning. For this purpose we propose an abductive version of the event calculus. The event calculus is a formalism for reasoning about events (or actions) and change formulated by Kowalski and Sergot [18]. Since its publication a number of abductive variants of it have been proposed in the planning and abduction literature [23, 25, 26]. Our formulation is a novel variant, in part inspired by the $\mathcal{E}$-language [16], to allow situated, resource-bounded agents to generate partial plans in a dynamic environment, possibly inhabited by other agents. Partial planning is useful for two reasons. Firstly, it allows the agents to interleave planning, sensing and acting. Secondly, it prevents agents from spending time and effort constructing complete plans that may become unnecessary or unfeasible when they get round to executing them.

4.1 An Abductive Formulation of the Event Calculus

We model a planning problem within the framework of the event calculus (EC) in terms of a (non-allowed) abductive logic program with constraints $KB_{plan} = \langle P_{plan}, I_{plan}, A_{plan} \rangle$. In a nutshell, the EC allows us to write meta-logic programs which "talk" about object-level concepts of *fluents*, *actions*, and *time points*. The main meta-predicates of the formalism are: $holds(F, T)$ (fluent

F holds at time T), *clipped*(T_1, F, T_2) (fluent F is clipped —from holding to not holding— between times T_1 and T_2), *declipped*(T_1, F, T_2) (fluent F is declipped —from not holding to holding— between times T_1 and T_2), *initially*(F) (fluent F holds from the initial time, say time 0), *happens*(A, T) (action A occurs at time T), *initiates*(A, T, F) (fluent F starts to hold after action A at time T) and *terminates*(A, T, F) (fluent F ceases to hold after action A at time T). Roughly speaking, in a planning setting the last two predicates represent the cause-and-effects links between actions and fluents in the modelled world. We will also use a meta-predicate *precondition*(A, F) (the fluent F is one of the preconditions for the executability of action A). In our KB_{plan}, we allow fluents to be positive (F) or negative ($\neg F$). Our formulation of the EC also contains the predicates *observed*/2, *executed*/2, *time_now*/2, *assume_holds*/2, and *assume_happens*/2, which will be described shortly.

We now define KB_{plan}. To this end we first show the KB_{plan} that would be used for complete planning (but by situated agents, interacting with their environment) and then show how it can be modified to allow for partial planning. P_{plan} consists of three parts: *domain-independent rules*, *domain-dependent rules*, and a *narrative part*. I_{plan} consists of *domain-independent integrity constraints* and possibly *domain-dependent integrity constraints*. The basic rules and integrity constraints for the domain-independent part, adapted from the original EC, are shown in Tables 2 and 3. The only abducible predicate here is *assume_happens*. The reason why we do not use the *happens* predicate as an abducible and instead define it in terms of *assume_happens* will become clear when we describe the (domain-independent) bridging rules given in Table 4. The first of the integrity constraints given in Table 3 states that a fluent and its negation cannot hold at the same time, while the second one expresses that when assuming (planning) that some action will happen, we need to enforce that each of its preconditions hold. The third constraint ensures that the actions in the resulting plan that have not been executed yet are scheduled for the future.

The *domain-dependent rules* define the predicates *initiates*, *terminates*, and *precondition* (as well as any other predicates required by the modelling of the concrete domain). An example is given in Table 6 (see also Section 4.3). The first two rules state that catching a train or driving to a destination X initiates being at X. The two *initially*-facts state that initially A has petrol but does not have any anti-freeze. The last integrity constraint has the flavour of a *reactive rule* and it states that if you are planning to drive somewhere and you have observed shortly before that it is snowing then you must make sure that you have anti-freeze. The other rules and constraints are self-explanatory.

The *narrative part* of P_{plan} defines the predicates *initially*, *observed* and *executed*. For instance, *initially*$(at(bob, (1, 1)))$ expresses that agent *bob* is initially at location $(1, 1)$; *executed*$(go(bob, (1, 1), (2, 2), 3), 5)$ says that *bob* went from location $(1, 1)$ to location $(2, 2)$ at time 3, and this has been observed at time 5 by the agent who contains it in its P_{plan}; and *observed*$(at(jane, (2, 2)), 5)$ states that *jane* is observed to be at location $(2, 2)$ at time 5. The narrative part is not only domain-dependent, but it also refers to a particular "running scenario", in

$holds(F, T_2)$	$\leftarrow$ $observed(F, T_1)$, *not* $clipped(T_1, F, T_2)$, $T_1 \leq T_2$
$holds(\neg F, T_2)$	$\leftarrow$ $observed(\neg F, T_1)$, *not* $declipped(T_1, F, T_2)$, $T_1 \leq T_2$
$clipped(T_1, F, T_2)$	$\leftarrow$ $observed(\neg F, T)$, $T_1 \leq T < T_2$
$declipped(T_1, F, T_2)$	$\leftarrow$ $observed(F, T)$, $T_1 \leq T < T_2$
$happens(A, T)$	$\leftarrow$ $observed(A, T)$
$happens(A, T)$	$\leftarrow$ $executed(A, T)$

Table 4. Domain-independent bridging rules in P_{plan}

$holds(F, T)$, $assume\_holds(\neg F, T) \Rightarrow$ *false*
$assume\_holds(F, T)$, $holds(\neg F, T) \Rightarrow$ *false*
$assume\_happens(A, T)$, $precondition(A, P) \Rightarrow assume\_holds(P, T)$

Table 5. Domain-independent integrity constraints in I_{plan} (for partial planning)

some concrete circumstances. Typically, *executed* facts within the narrative part of KB_{plan} of an agent refer to actions executed by the agent, whereas *observed* facts refer to properties of the environment, via facts of the form $observed(L, T)$ (the fluent literal L has been observed to hold at time T) as well as actions executed by other agents, via facts of the form $observed(A, T)$ (the action A has been observed to have happened at time T). The parameters of A can contain the identification of the agent who has executed the action. To accommodate observations, we add the *bridging rules* shown in Table 4 to P_{plan}. Note that the narrative part of P_{plan} changes over the life time of the agent (whereas the other parts of the knowledge base remain fixed).

Planning is done by reasoning with P_{plan} and I_{plan} to generate appropriate instances of the abducible predicate *assume_happens*, the successful execution of whose actions would establish the planning goal. Note that we need such a predicate (and *happens* cannot be used directly), because abducible predicates cannot be defined in the theory.

Now we show how KB_{plan} can be modified to facilitate partial planning. A partial plan contains subgoals as well as actions. Such subgoals are modelled using an additional abducible predicate *assume_holds*. Table 5 lists the additional integrity constraints in KB_{plan} pertaining, specifically, to partial planning. The purpose of the first two of the additional integrity constraints is to make sure that no fluent is assumed to hold at a time when the contrary of the fluent holds. The final constraint replaces the second constraint in Table 3.

This formulation allows the agent to plan for its goals by generating actions and subgoals that correspond to the preconditions of the actions, all of which are consistent with one another and with the observations that have been made. (An alternative formulation for partial planning can add the integrity constraints of Table 5 without modifying those in Table 3, but instead adding the following rule to P_{plan}: $holds(F, T) \leftarrow assume\_holds(F, T)$. This would allow a more "liberal" way of partial planning, but for computational reasons we have chosen the first approach in the implementation.)

To summarise, our abductive formulation of the EC for partial planning consists of $KB_{plan} = \langle P_{plan}, I_{plan}, A_{plan} \rangle$, where

- P_{plan} consists of the rules in tables 2 and 4 and any *domain-dependent rules*,
- I_{plan} consists of the integrity constraints in Table 5 and the first and last constraints in Table 3 and any *domain-dependent integrity constraints*, and
- A_{plan} consists of the predicates *assume_happens* and *assume_holds*.

Intuitively, our proposal adds to more conventional abductive EC theories for planning the possibility to interact with the environment, by observing that properties hold and that other agents have executed actions. These additions pave the way to the situatedness of the agent in the environment, when planning is performed within a *"sense-plan-act"* cycle. Indeed, $observed(L, T)$ and $observed(A, T)$ facts (where A is an action executed by some other agent) are added to the narrative part of KB_{plan} as the result of a sensing operation of the agent, whereas $executed(A, T)$ facts (where A is an action executed by the agent) are added as the result of the execution of a planned action in the environment. Each step in the cycle takes place at some concrete time, used to time-stamp the facts recorded in the narrative part of KB_{plan}.

4.2 Goals and Partial Plans in CIFF

Here we describe in more detail how goals and partial plans of an agent are specified within the framework of the EC. A *goal* is a conjunction of the form $holds(L, T) \wedge TC$, where TC is a set of constraints on T, referred to as the *temporal constraints* of the goal. A *(partial) plan* for a given goal consists of

- a (possibly empty) set of atoms of the form $assume\_happens(A, T) \wedge TC$, where TC is a set of constraints on T, referred to as the *Actions* in the plan;
- a (possibly empty) set of atoms of the form $assume\_holds(L, T) \wedge TC$, where TC is a set of constraints on T, referred to as the *SubGoals* in the plan.

A partial plan for a set of goals is a set of partial plans, one for each individual goal. Goals, actions, subgoals and temporal constraints will be typically non-ground, and the variables occurring in them are implicitly existentially quantified over the set of (partial) plans and goals.

Given a set of goals, GS, where each goal is of the form $holds(L, T) \wedge TC$, a (possibly empty) set of subgoals *SubGoals*, a (possibly empty) set of actions *Actions* (representing already existing partial plans that we are trying to expand), and a (possibly empty) set of temporal constraints, to compute a partial plan for GS at a time τ, CIFF uses the program $\langle Th, I \rangle$, where Th is a set of iff-definitions formed by the selective completion of P_{plan}, given A_{plan}, and I is the set of all integrity constraints in I_{plan}.

To choose a query for CIFF, we first have to select the goals to be planned for in the next round of planning. Let *gs(Goals, Time)* be a goal *selection function* that takes as input a set of goals *Goals* of the form $holds(L, T) \wedge TC$ and a time of evaluation *Time*, and returns as output a subset of *Goals*. We do not give a definition for such a selection function here. A number of different definitions

$initiates(catch\_train\_to(X), T, at(X))$
$initiates(drive\_to(X), T, at(X))$
$initiates(fill\_up(X), T, have(X)) \leftarrow X = petrol$
$initiates(fill\_up(X), T, have(X)) \leftarrow X = anti\_freeze$
$initially(\neg have(anti\_freeze))$
$initially(have(petrol))$
$precondition(drive\_to(X), have(petrol))$
$assume\_happens(catch\_train\_to(X), T),\ holds(train\_strike, T) \Rightarrow false$
$assume\_happens(drive\_to(X), T_1),\ observed(snowing, T_2),\ T_1 - 5 \leq T_2 \leq T_1 \Rightarrow$
$assume\_holds(have(anti\_freeze, T_1))$

Table 6. Domain-dependent rules in KB_{plan} for the airport example

are possible, for example *gs* can select those goals that are deemed "urgent" according to some measure of urgency (e.g. how close their times are to *Time*). Let $GS' = GS \cup \{holds(L,T) \wedge TC \mid assume\_holds(L,T) \wedge TC \in SubGoals\}$. Let $gs(GS', \tau) = SelectedGoals$. CIFF is then invoked with a query including:

(1) all selected goals: the conjunction of all atoms $holds(L,T) \wedge TC$, for all the goals in *SelectedGoals*;
(2) all non-selected (sub)goals: the conjunction of all $assume\_holds(L',T') \wedge TC$ such that $holds(L',T') \wedge TC$ is in GS or $assume\_holds(L',T') \wedge TC$ is in *SubGoals*, and $holds(L',T') \wedge TC$ is not in *SelectedGoals*;
(3) $time\_now(\tau)$;
(4) the conjunction of all atoms $assume\_happens(A',T') \wedge TC$ in *Actions*.

4.3 Example

To illustrate the application of CIFF to planning consider the following scenario. Agent A has the goal of being at the airport before time 10. It knows of two ways of getting there, by train and by car. It has already learned that there is a train strike. So it plans to drive there. Then it makes two observations, one that it is low on petrol, and, the other, that it is snowing. So it plans to get petrol to accommodate the first observation, and knowing that it is short on anti-freeze it reacts to the information about the snow by adding a goal of topping up its anti-freeze (thus repairing its plan to fit in with its changed environment). So through the cycle of observations and (partial) planning it finally constructs a complete plan consisting of the three actions of filling up petrol, topping up anti-freeze, and driving to the airport at appropriate times. The domain-dependent and narrative parts of agent A's KB_{plan} are shown in Table 6.

Suppose we are now at time 4 and we have the goal of being at the airport before time 10. The goal given to CIFF would be the following:

$$holds(at(airport), T),\ T < 10,\ time\_now(4)$$

CIFF will return the following partial plan (with $4 < T' < 10$):

$$assume\_holds(have(petrol), T'),\ assume\_happens(drive\_to(airport), T')$$

At this stage, now at time 5, suppose we make the two observations $observed(\neg have(petrol), 5)$ and $observed(snowing, 5)$. These are recorded in agent A's P_{plan}. For the next round of planning, at time 6, say, CIFF can be called with the following set of goals:

$$holds(have(petrol), T'),\ assume\_happens(drive\_to(airport), T'),\\ 4 < T',\ T' < 10,\ time\_now(6)$$

CIFF will then augment the existing partial plan by adding to it the following new subgoals and actions (with the additional constraints $6 < T''$ and $T'' < T'$):

$$assume\_holds(have(anti\_freeze), T'),\ assume\_happens(fill\_up(petrol), T'')$$

At time 7, say, if there are no further observations, CIFF will complete the plan by adding the action $fill\_up(anti\_freeze)$ with appropriate time constraints.

4.4 Implementation of the Planner

Given the computational model for planning put forward above and our implementation of the CIFF proof procedure described in Section 3, the implementation of a simple abductive planner has been straightforward. In essence, it consists of only a single Prolog clause:

```
plan( Narration, Assumptions, Goals, TCs, Answer) :-
  kbplan( PlanDefs, ICs),
  close_pred( executed/2, Narration, ExecActions),
  close_pred( observed/2, Narration, Observations),
  close_pred( time_now/1, Narration, TimeNow),
  Defs = [ExecActions,Observations,TimeNow|PlanDefs],
  append( Goals, TCs, Goals1), append( Goals1, Assumptions, Query),
  ciff( Defs, ICs, Query, Plan:Substitution:NewTCs),
  delete_list( Plan, Assumptions, NewPlan),
  Answer = NewPlan:Substitution:NewTCs.
```

`Narration` is a list of (ground) terms of the form `executed(Action,T)`, `observed(Fluent,T)`, and `observed(Action,T)`, representing the narrative part of KB_{plan}, as well as a single term of the form `time_now(N)` to communicate the current time (`N` is an integer). `Assumptions` is a list of terms of the form `assume_holds(Goal,T)` and `assume_happens(Action,T)` encoding the goals and actions in the current partial plan. `Goals`, the goals to plan for, is a list of terms of the form `holds(Goal,T)` and `TCs` is a list of temporal constraints over variables occurring in the goals and actions given in the input. The variable `Answer` will be instantiated with a representation of the chosen plan if there exists one; otherwise the call to `plan/5` will fail.

In the first subgoal of the implementation of `plan/5`, the `kbplan/2` predicate is used to retrieve the iff-definitions (`PlanDefs`) and the integrity constraints (`ICs`) in the non-narrative parts of KB_{plan} (we assume these have been asserted earlier). The predicate `close_pred/3` is used to generated iff-definitions for the

predicates occurring in the `Narration` and these are appended to the list of definitions to obtain `Defs`. Then the CIFF proof procedure is called with `Defs` as the background theory, `ICs` as the integrity constraints, and the list of all other relevant terms as the query. The first component of the answer consists of a list of abducible predicates encoding the plan (using `assume_holds/2` for subgoals and `assume_happens/2` for actions). To simplify the output, the assumptions (goals and actions) already present in the input are then deleted from this list. Furthermore, the answer may also include a list of restrictions on the answer substitution (`Substitution`) and an updated list of constraints (`NewTCs`).

5 Conclusion

We have presented a Prolog implementation of the CIFF procedure that extends the general purpose abductive proof procedure IFF by dealing with non-allowed programs and by handling constraints. The implementation allows the use of any abductive logic program and presents answers in the form of abducibles with instantiations and restrictions of variables. We have also discussed an application to planning where, by varying the input theory, we can construct complete or partial plans in the presence or absence of narrative information.

The CIFF system has been used extensively in the PROSOCS framework [27] to implement a planning component as well as for reactivity and temporal reasoning. Reactivity allows condition-action rule behaviour used in PROSOCS, for example, for plan repair, for strategies for negotiation and communication with other agents, and generally for reacting to changes in the environment. The temporal reasoning capability is based on an extensive abductive logic program based on the event calculus that deals with the revision of the agent's knowledge as a result of assimilating new information from its environment, including other agents [2]. We have been able to use and test the CIFF system on a number of examples for all three applications without any modifications. While so far, we have concentrated on providing an implementation of CIFF that correctly implements the semantics, that is easy to understand, and that supports future extensions, in our future work we hope to also study possible optimisation techniques for CIFF.

Acknowledgements. This work was supported by the European Commission FET Global Computing Initiative, within the SOCS project (IST-2001-32530). We thank all SOCS partners for useful feedback and in particular Marco Gavanelli and Michela Milano for suggestions on implementing the constraint solver.

References

1. B. Beckert and J. Posegga. leanTAP: Lean, Tableau-based deduction. *J. Automated Reasoning*, 15(3):339–358, 1995.
2. A. Bracciali and A. C. Kakas. Frame consistency: Computing with causal explanations. In *Proc. NMR-2004*, 2004. To appear.

3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. PLILP-97*, 1997.
4. K. L. Clark. Negation as failure. In *Logic and Data Bases*. Plenum Press, 1978.
5. L. Console, D. T. Dupré, and P. Torasso. On the relationship between abduction and deduction. *J. Log. Computat.*, 1(5):661–690, 1991.
6. M. Denecker and D. De Schreye. SLDNFA: An abductive procedure for abductive logic programs. *J. Log. Prog.*, 34(1):111–167, 1998.
7. M. Denecker and B. Van Nuffelen. Experiments for integration CLP and abduction. In *Proc. Workshop on Constraints*, 1999.
8. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure: Definition and soundness results. Technical Report 2004/2, Department of Computing, Imperial College London, 2004.
9. K. Eshghi and R. A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proc. ICLP-1989*. MIT Press, 1989.
10. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *J. Log. Prog.*, 33(2):151–165, 1997.
11. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Prog.*, 19-20:503–582, 1994.
12. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.
13. A. C. Kakas and P. Mancarella. Abductive logic programming. In *Proc. Workshop Logic Programming and Non-Monotonic Logic*, 1990.
14. A. C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proc. ECAI-2004*, 2004. To appear.
15. A. C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive constraint logic programming. *J. Log. Prog.*, 44:129–177, 2000.
16. A. C. Kakas and R. Miller. A simple declarative language for describing narratives with actions. *J. Log. Prog.*, 31(1–3):157–200, 1997.
17. A. C. Kakas, B. Van Nuffelen, and M. Denecker. A-system: Problem solving through abduction. In *Proc. IJCAI-2001*. Morgan Kaufmann, 2001.
18. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
19. R. A. Kowalski, F. Toni, and G. Wetzel. Executing suspended logic programs. *Fundamenta Informaticae*, 34:203–224, 1998.
20. K. Kunen. Negation in logic programming. *J. Log. Prog.*, 4:289–308, 1987.
21. F. Lin and J.-H. You. Abduction in logic programming: A new definition and an abductive procedure based on rewriting. *Artif. Intell.*, 140(1/2):175–205, 2002.
22. P. Mancarella and G. Terreni. An abductive proof procedure handling active rules. In *Proc. AI*IA-2003*. Springer-Verlag, 2003.
23. L. Missiaen, M. Bruynooghe, and M. Denecker. CHICA, an abductive planning system based on event calculus. *J. Log. Computat.*, 5(5):579–602, 1995.
24. F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In *Proc. AI*IA-1999*. Springer-Verlag, 2000.
25. M. P. Shanahan. Prediction is deduction but explanation is abduction. In *Proc. IJCAI-1989*. Morgan Kaufmann, 1989.
26. M. P. Shanahan. *Solving the Frame Problem*. MIT Press, 1997.
27. K. Stathis, A. C. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: A platform for programming software agents in computational logic. In *Proc. AT2AI-2004*, 2004.
28. Swedish Institute of Computer Science. *Sicstus Prolog User Manual*, 2003.