ACLP: ABDUCTIVE CONSTRAINT LOGIC PROGRAMMING

A.C. Kakas, A. Michael, C. Mourlas

▷ This paper presents the framework of Abductive Constraint Logic Programming (ACLP), which integrates Abductive Logic Programming (ALP) and Constraint Logic Programming (CLP). In ACLP, the task of abduction is supported and enhanced by its non-trivial integration with constraint solving. This integration of constraint solving into abductive reasoning facilitates a general form of constructive abduction and enables the application of abduction to computationally demanding problems.

The paper studies the formal declarative and operational semantics of the ACLP framework together with its application to various problems. The general characteristics of the computation of ACLP and of its application to problems are also discussed. Empirical results based on an implementation of the ACLP framework on top of the CLP language of ECLiPSe show that ACLP is computationally viable, with performance comparable to the underlying CLP framework on which it is built. In addition, our experiments show the natural ability for ACLP to accommodate easily and in a robust way new or changing requirements of the original problem. ACLP thus combines the advantages of modularity and flexibility of the high-level representation afforded by abduction together with the computational effectiveness of low-level specialised constraint solving. ◁

1. INTRODUCTION

Abduction enhances the expressivity of a representation framework allowing a high-level representation of problems close to their natural specification. It has been shown to be suitable for capturing different kinds of inferences useful in Artificial

Address correspondence to Department of Computer Science, University of Cyprus, CY-1678 Nicosia, Cyprus.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

Intelligence (AI) such as explanation, non-monotonic and default reasoning and theory revision. As a result, abduction manifests itself as an important form of inference for addressing a variety of problems in AI (see e.g. [1], [4], [5], [7], [13], [27], [33], [46], [23], [42] [51], [52]). These problems include, amongst others, the problems of diagnosis, planning and scheduling, natural language understanding, assimilation of sensor data and user modeling. The high-level modeling environment of abduction can offer an extra layer of modularity and flexibility useful for these applications with complex and changing requirements. Although the utility of abduction for formulating these problems is well proven there has been little work (see though [42]) to address the question of whether these abductive formulations can form the basis for computationally effective solutions to realistic problems.

Abductive applications are based on the view that the required goals to be solved can be seen as observations to be explained by abduction. Abductive explanations constitute solutions to these goals. In many applications, it is often useful (or indeed necessary) to allow non-ground abducible assumptions of the form $\exists X a(X)$ where these variables are restricted through some set C of constraints defined over an (arithmetic) domain. For example, in abductive planning we need to assume that actions occur at some times which are not necessarily fixed but instead must satisfy certain time constraints amongst them. It then transpires that one way to improve the effectiveness of abductive solutions is to use together with abduction, a specialized constraint solver for handling such constraints in cooperation with abductive reasoning.

The aim of this work is to develop a new framework for abduction, with a firm semantic basis, in which it is possible to compute abductive solutions by interfacing constraint solving to abduction. This new framework, called Abductive Constraint Logic Programming (ACLP), is the result of the integration of the two frameworks of Abductive Logic Programming (ALP) [7, 23] and Constraint Logic Programming (CLP) [19, 20, 21, 57].

In ACLP, the task of abduction is supported and enhanced by its non-trivial integration with constraint solving. This framework facilitates a more general form of constructive abduction and enables the application of abduction to problems that would otherwise be computationally difficult to solve. In the computation of ACLP, abductive reasoning provides an incremental reduction of the high-level problem representation and goals to abductive hypotheses and lower-level constraints, whose form is problem independent. The integration of abductive reasoning with constraint solving in ACLP is non-trivial, in the sense that the constraint solver not only solves the final constraint store generated by the abductive reduction but also affects dynamically the abductive search for a solution. It enables abductive reductions to be pruned early by setting new suitable constraints on the abducible assumptions into the constraint store, provided that this remains satisfiable. During the ACLP computation there is a non-trivial cooperation between (i) reduction of goals and consistency checking of abducible assumptions, (ii) setting new constraints in the constraint store of reduction and (iii) generating further abductive hypotheses.

The paper presents the formal declarative and operational semantics of the ACLP framework and studies how this can be used for the applications of planning and scheduling. The general characteristics of the abductive computation in ACLP are highlighted. Similarly, we discuss the general features of the use of ACLP to applications based on the experience of the problems developed so far under this framework. The paper brings together and extends earlier work on the ACLP framework [30, 32]. It extends this previous work both in the theory and the practical application of ACLP.

In order to examine more closely the properties of the ACLP framework we have performed two types of experiments using an implementation of ACLP on top of the CLP language of ECLiPSe [11]. The first group of experiments aims to test the computational viability of the ACLP system and its performance, in comparison with the direct use of the (lower-level) constraint solving framework of CLP on which ACLP is built. The other type of experiments aims to investigate the high-level expressivity of ACLP and its ability to accommodate in a natural way changes in the problem definition. These experiments provide evidence that the framework of ACLP does not compromise significantly the computational efficiency of the underlying CLP framework while it offers a high degree of flexibility in its representation of problems.

The conceptual similarity between ALP and CLP has motivated a number of other proposals to integrate the treatment of abducibles and constraints e.g. [7, 15, 37]. In most of these works the primary aim is to unify the two concepts into one, by translating them into each other. In our work we take a different approach where the emphasis is to strongly couple abduction and constraint solving while preserving their separate identities. Features from each framework are to be used directly, aiming to exploit their respective advantages: the high-level expressiveness of ALP and the constraint solving capabilities of CLP.

The rest of this paper is organized as follows. Section 2 defines the semantical framework of ACLP and formalizes its integrated model of computation by a new proof procedure for ACLP. This proof procedure is used as a basis for the implementation of the ACLP system and its application to the problems of planning and scheduling in section 3. In the same section, the overall computation of ACLP is illustrated and experimental results on the high-level expressivity, the flexibility and the computational viability of the ACLP system are presented. The general features and characteristics of using ACLP in applications are presented in section 4. Section 5, reports on other related work and section 6 concludes together with a discussion on future work.

The ACLP system together with some documentation and example programs can be found at http://www.cs.ucy.ac.cy/aclp/.

2. THE ACLP FRAMEWORK

In this section we present the formal declarative and operational semantics of the Abductive Constraint Logic Programming ($ACLP$) framework. In this framework abduction is supported by the constraint solving capabilities of CLP. From a formal point of view, $ACLP$ can be seen as an extension of the ALP framework that supports constructive abduction allowing the abducible hypotheses to take the non-ground form of $\exists X(A(X), C(X))$, where A is a conjunction of abducible atoms and C is a set of constraints defined over the CLP (arithmetic) domain.

2.1. Preliminaries

We will assume some familiarity with basic concepts of logic programming as found in [38]. We present here some of the basic preliminary material from the theory of CLP on which the $ACLP$ framework builds. The required material from ALP is developed as needed in the section. Further details on ALP and CLP can be found in the recent surveys of [23] and [22] respectively.

We will adopt the following notation. Variables (or sets of variables) will be denoted by X,Y,Z,.... Terms (or sets of terms) are denoted by u,t,.... Atoms (or sets of atoms) are denoted by A,B or L. Also p(X),q(X),... denote atoms whose predicate symbol is p,q,... and contain the variable (or set of variables) X. All symbols may be subscript.

A Constraint Logic Programming framework, CLP($\Re$) is defined over a particular structure $\Re$. $\Re$ consists of domains[1] $D\Re$ and a set Π_c of constraint symbols Π_c which includes equality, together with an assignment of relations on $D\Re$ for each such symbol in Π_c that respects its arity. A constraint takes the form $c(u_1, \ldots, u_n)$ where c is an n-ary predicate symbol in Π_c and u_i are terms of $\Re$. As usual, a term can be a variable, a constant that names an element of the domain $D\Re$, or $f(t_1, \ldots, t_k)$, where f is a function from $D\Re^k$ to $D\Re$ and $t_1, \ldots, t_k$ are terms. Together with the constraint predicate symbols in Π_c we also have a set of (user-defined) predicate symbols Π_p which is disjoint from Π_c.

A constraint logic program P, in CLP($\Re$), consists of a finite set of rules, each being of the form

$$p_0(t_0) \leftarrow c_1(u_1), \ldots, c_n(u_n), p_1(t_1), \ldots, p_m(t_m)$$

where p_i are (user-defined) predicate symbols in Π_p, different from the constraint predicate symbols Π_c in $\Re$, c_i are constraints in $\Re$ and u_i,t_i are terms of $\Re$. A rule will also be denoted by, $A \leftarrow C[]A_1, \ldots, A_n$, where C is a set of constraints and A, $A_1, \ldots, A_n$ are atoms with predicate sympols in Π_p.

A *goal*, G, has the same form as the body of a program rule but whose variables are as usual understood to be existentially quantified. Sometimes we will also write G as $\leftarrow G$.

Definition 2.1. (Valuation/Solvability)
A valuation θ of a set of variables is a mapping from these variables to $D\Re$. A valuation θ, on the set of all variables appearing in a set of constraints C, is called an $\Re$-solution of C iff C_θ, obtained by applying θ to C, is satisfied in $\Re$. The set C is called $\Re$-solvable (or consistent) iff it has at least one $\Re$-solution.

The meaning of a constraint logic program P is given by the $\Re$-model of the program. To define this we consider the $\Re$-base of a program P, given by the set
$\{p(X_1, X_2, ..., X_n)\theta :$
p is an n-ary symbol in Π_p and
θ is a valuation on the variables $X_1, X_2, \ldots, X_n\}$.

Definition 2.2. ($\Re$-model)
An $\Re$-model of a program P is given by a subset I of the $\Re$-base such that for every rule, $A \leftarrow C[]A_1, \ldots, A_n$, where $n \geq 0$, in P and for every valuation θ,

[1] In general, we may have a many sorted structure with a domain for each sort. For simplicity of presentation we assume that we have only one sort.

such that θ is an $\Re$-solution of C we have $\{A_1\theta, \ldots, A_n\theta\} \subseteq I \Rightarrow A\theta \in I$.
The $\Re$-model of P is the least $\Re$-model, denoted by $M_P^{\Re}$, of P.

We will denote by $\models_{\Re}$ (or simply $\models$) truth in the model $M_P^{\Re}$ together with solvability in $\Re$ over valuations in $\Re$. eg. $M_P^{\Re} \models G \wedge C$ means that there exists a valuation θ of the variables in G and C, such that C_θ is solvable in $\Re$ and G_θ is true in $M_P^{\Re}$. For $M_P^{\Re} \models_{\Re} G$ we will also write $P \models_{\Re} G$.

Definition 2.3. ((P,$\Re$)-derivation)
A (P,$\Re$)-derivation of a goal G in a program P is a sequence of goals, starting from G, where each goal G_i in the sequence is obtained by resolving the previous goal with a rule in the program P (on a non-constraint atom) such that the resulting set of constraints, that appears in G_i, is $\Re$-solvable.

A successful derivation from a goal G is a finite (P,$\Re$)-derivation whose last element is a goal with no atoms. Finite derivations from G which do not satisfy this property are finitely failed (FF). Associated with a successful derivation of a goal G is an answer constraint C, this being the constraint in the last goal of the derivation. The following soundness theorems are proved in [18, 21].

Theorem 2.1. [18, 21]
If G has a successful (P,$\Re$)-derivation with answer constraint C then

$$M_p^{\Re} \models G \leftarrow C$$

Theorem 2.2. [18, 21]
If all (P,$\Re$)-derivations for a goal G are finitely failed then

$$M_p^{\Re} \models \neg G$$

2.2. ACLP Semantics

The integration of the frameworks of ALP and CLP is based on the view that they can be both understood within the same conceptual framework of hypothetical reasoning. The satisfaction of a goal, in either framework, is understood conditionally on a set of hypotheses, abducible assumptions for ALP or constraints for CLP. This uniform view of the two frameworks facilitates their integration.

Given an underlying framework of CLP($\Re$), an abductive theory in ACLP is defined as follows.

Definition 2.4. (Abductive CLP theory)
An abductive CLP($\Re$) (or ACLP) theory is a triple $\langle P, A, IC\rangle$ where:

- *P :* is a constraint logic program.
- *A :* is a set of abducible predicates different from the constraint predicates.
- *IC:* is a set of closed first order formulae (Integrity Constraints) over the combined language of CLP($\Re$) and P.

An ACLP theory thus contains three types of predicates: (i) ordinary predicates as in standard LP, (ii) constraint predicates as in CLP and (iii) abducible predicates as in ALP. The abducible predicates are normally not defined in the program and any knowledge about them is represented either explicitly or implicitly in the integrity constraints IC. We will assume that the program P does not contain any rules for abducible predicates.

The *abducibles* of an ACLP framework, from which the answers will be build, are given by the next definition.

Definition 2.5. (Abducibles)
Let $\langle P, A, IC\rangle$ be an abductive CLP($\Re$) theory. Then the set of abducibles, also called abductive hypotheses or assumptions, consists of the following formulae :

1. a(d), where $a \in A$ & d is a vector of constants in D$\Re$
2. $\exists X(a_1(X), .., a_n(X), C(X))$, where $n \geq 1$, $a_i \in A$, $\forall i. 1 \leq i \leq n$ and $C(X)$ is a (possibly empty) set of constraints.

Abducibles in 1 are called ground abducibles.

Answers to goals will have the form of a subset of abducibles Δ together with a set of constraints of the underlined CLP theory. The correctness of an answer is given through the notion of an abductive explanation for G.

The definition of general abductive explanations, that can include also non-ground abducibles e.g. $\Delta = \exists X(a(X), C(X))$, is based on the definition of ground explanations, by considering the different possible groundings of Δ allowed by the contraints C that it involves.

Definition 2.6. (Grounding of Δ)
Let $\langle P, A, IC\rangle$ be an abductive CLP($\Re$) theory and Δ be a set of abducibles. If Δ is ground then the grounding of Δ is itself. Otherwise, if Δ is not ground, then a grounding of Δ is a set of ground abducibles Δ_g obtained by applying to each non-ground abducible, $\exists X(a_1(X), .., a_n(X), C(X))$, that appears in Δ a valuation g on the variables X that satisfies the constraints $C(X)$.

The valuation g is called a grounding (or grounding valuation) of Δ.

Definition 2.7. (Grounding of a Goal)
Let G be a goal. If G is ground then the grounding of G is itself. Otherwise, if G contains free variables, then a grounding of G, denoted by G_g is the goal obtained from G by applying a valuation g on a subset of the variables of G.

Definition 2.8. (Ground abductive explanation)
Let $\langle P, A, IC\rangle$ be an abductive CLP($\Re$) theory. Then, a ground abductive explanation of a goal G is a set, Δ_g, of ground abducibles such that there exists a solvable set C of constraints and a grounding G_g of G with

$$M^{\Re}_{P\cup\Delta_g} \models_{\Re} \forall(G_g \leftarrow C) \text{ and}$$

$$M^{\Re}_{P\cup\Delta_g} \models_{\Re} IC,$$

where $M^{\Re}_{P\cup\Delta_g}$ is the minimal $\Re$-model of the program $P \cup \Delta_g$ obtained by augmenting P with the set of facts in Δ_g. As in CLP the set C is called an answer constraint for G.

Definition 2.9. (Abductive explanation)
Let $\langle P, A, IC \rangle$ be an abductive CLP($\Re$) theory. Then an abductive explanation of a goal G is a set Δ of abducibles such that

1. there exists a valuation (or grounding) g such that, for each abducible formula ϕ in Δ, the constraints in ϕ are satisfied or equivalently, there exists at least one ground set of abducibles, Δ_g, such that $\Delta_g \models_\Re \Delta$ and
2. for any grounding Δ_g of Δ, Δ_g is a ground abductive explanation of G.

Hence Δ is an abductive explanation for a goal G iff there exists (in $\Re$) at least one consistent grounding of Δ and for any such grounding, Δ_g, this constitutes a ground abductive explanation of G.

Note that an answer to a goal G consists of an abductive explanation Δ together with a set C of additional CLP constraints referred to in definition 2.8. Hence an answer contains two separate parts of CLP constraints: those which constrain the abducible assumptions and appear together with the abducibles in Δ, and those which are on the free variables of the goal G but are not related to any of the abducible assumptions. The later are the additional constraints C that are needed in definition 2.8.

The semantics of the integrity constraints, as formalised by the above definition, is given by their truth in the intended computed model of $P \cup \Delta_g$. The integrity constraints are therefore formalised, following the proposals of [45, 24, 34], as meta-level properties that must hold true in the abductive extension $P \cup \Delta_g$ of the program. It will be useful to have the following definition.

Definition 2.10. (Satisfaction of Integrity Constraints)
Let $\langle P, A, IC \rangle$ be an abductive CLP($\Re$) theory and let Δ_g be a set of ground abducibles. We say that $M^\Re_{P\cup\Delta g}$ or Δ_g satisfies the integrity constraints IC (or is consistent) whenever $M^\Re_{P\cup\Delta g} \models_\Re IC$.

We will always assume that for any given ACLP theory $\langle P, A, IC \rangle$ this is initially consistent i.e. that the integrity constraints are satisfied by the program P alone.
At this semantic level we will allow any closed first order formula to be an integrity constraint. Later in section 2.3, which studies the operational model of the framework, the integrity constraints will be restricted to a special form.

To illustrate the above definitions consider the following example, where the domain of $\Re$ is the integers and Π_c contains the equality and inequality constraint predicates.

Example 2.1.
$P: \quad p(X, Y) \leftarrow X > 3, q(X, Y), a(X).$
$\qquad\; q(X, Y) \leftarrow X \leq 5, Y \geq 2.$

where the only abducible predicate is “a”.

The goal G=$p(X,Y)$, i.e G=$\exists X,Y p(X,Y)$, has $\Delta = \exists X(a(X), X > 3, X \leq 5)$ as an abductive explanation since $\Delta_g := a(5)$ and $\Delta_g := a(4)$ are ground explanations of the goal. The goal can be satisfied in $P \cup \Delta_g$ with the constraint $C = \{Y \geq 2\}$. If we add the integrity constraint ic: $\leftarrow X > 4, a(X)$[2], which is a denial stating that we cannot have $a(X)$ for any $X > 4$, then the above Δ is not an explanation of G. This is because although $X = 5$ satisfies the constraints in Δ, $\Delta_g = a(5)$ is not a ground explanation of G any more. An explanation of G that satisfies the integrity constraint is $\Delta' = \exists X(a(X), X > 3, X \leq 5, X \leq 4)$ which is equivalent to $\Delta'' = \exists X(a(X), X = 4) \equiv a(4)$.

Notice further that in each case Δ_g is an explanation of a corresponding goal G_g obtained from G= $p(X,Y)$ by grounding the variable X in G with the same constant as that used to ground X in Δ to get Δ_g, ie. $\Delta_g = a(4)$ is an explanation of $p(4,Y)$ and $\Delta_g = a(5)$ is an explanation of $p(5,Y)$. We will see below that when we compute an abductive explanation Δ of a goal G the associated answer constraint C that would be generated by the ACLP computation will define for each Δ_g a corresponding (partially) ground goal G_g.

2.3. ACLP Proof Procedure

We now present the operational semantics of the integrated framework of ACLP in terms of a proof procedure for solving/explaining goals. This procedure integrates the abductive proof procedures in [14, 26, 27] with the computational framework of CLP and exploits the constraint solving of CLP to support and enhance the computation of abductive solutions.

The computation in ACLP consists of two interleaving phases, called abductive and consistency phases. In the abductive phase, hypotheses on the abducible predicates are generated, by reducing the high-level goals, and added to a set of abductive assumptions Δ. The consistency phase checks whether these hypotheses are an allowed addition to the assumption set in the sense that the integrity constraints can remain satisfied by this addition. Together with this assumption set a constraint store (often also referred to as a global constraint store) C of CLP constraints is also generated. This constraint store can grow in both phases of the computation provided that it remains satisfiable throughout the computation. A constraint solver is used to decide on the satisfiability of this store when necessary. The satisfiability of C in turn affects back the overall abductive computation.

As in [26, 27] we will assume that the integrity constraints that can be handled *directly* by the ACLP proof procedure have a restricted form.

Definition 2.11. (Restricted Integrity Constraints)
An integrity constraint is of restricted ACLP form if it has the denial form $\leftarrow C[]\ p_1, \ldots, p_m$ where each p_j is a (positive) atom and at least one of these has an abducible predicate.

[2] We will often use a “programming” notation to write integrity constraints. For example, here “$\leftarrow X > 4, a(X)$” represents the integrity constraint $\neg\exists X\ (a(X) \wedge X > 4)$.

Note that integrity constraints of this restricted form are trivially satisfied by the program alone, when (as we have assumed) the abducibles do not have definitions in the program P.

Later in this section we will discuss how this restriction can be relaxed by transforming other constraints into this form. We first illustrate the procedure with an example.

Example 2.2.
P : $p(X) \leftarrow X \geq 2, q(X), a(X).$ IC $: \leftarrow Y \geq 6, a(Y), r(Y).$ A $= \{a\}$
$q(x) \leftarrow X > 4, X < 10$.
together with rules in P defining the predicate r.

To prove the goal $G_0 : \leftarrow p(X)$ we can resolve against the first and second clauses to obtain the new goal $G_1 : \leftarrow X \geq 2, X > 4, X < 10, a(X)$. We then proceed to abduce $\exists X a(X)$ by first naming this existential variable with a new variable n, called a naming variable, and adding $a(n)$ to the set of hypotheses Δ_0. We also extend the (global) constraint store $C_0 = \{X \geq 2, X > 4, X < 10\}$, which can be reduced to $C_0' = \{X > 4, X < 10\}$, with the naming constraint $X = n$ and then continue to check for the consistency of the assumption $a(n)$ with respect to the integrity constraints in IC.

For the consistency of $a(n)$, we resolve with the integrity constraint in IC to get the goal $\leftarrow Y \geq 6, Y = n, r(Y)$. This goal must fail. One way to fail is for the local constraints[3]in the goal to become unsatisfiable. This can be ensured by adding a (suitable) negation of these constraints to the global constraint store and then checking for the solvability of this extended store. In effect, this adds new constraints on the naming variable n associated with the abducible assumption $a(n)$. In this example, the negation of the local constraints $(Y \geq 6, Y = n)$ will be $\neg\exists Y(Y \geq 6, Y = n)$ which is equivalent to $(n < 6)$. The constraint store $C_1 = \{X > 4, X < 10, X = n\}$ is then extended to $C_2 = \{X > 4, X < 10, X = n, n < 6\}$ which is equivalent to $C_3 = \{n > 4, n < 6\}$. This gives a final solution to the goal G which after reverse naming, is the explanation $\Delta_{rn} = \exists Z(a(Z), Z > 4, Z < 6)$. Note that in general it is not necessary to add the negation of the (whole) constraint store, of the reduced integrity constraint, to the global constraint store. We can add any constraint that is sufficient to make this constraint store unsolvable provided that the global store remains solvable.

In this example a constraint solver for the CLP constraints is used to check the satisfiability of (also reducing) the various instances C_0, C_1 and C_2 of the global constraint store, and to help make the local set of constraints in the consistency phase unsatisfiable. In general, the constraint solver is essentially a black box, transparent to the abductive reasoning, that is consulted during the overall abductive computation with specific tasks of the following two types:

1. Check the satisfiability of and reduce (or solve) the constraint store at each step in the abductive phase. Similarly, in the consistency phase check the

[3]We use the term local constraints (or local constraint store) to refer to the set of constraints accumulated from the reduction of an integrity constraint in the consistency phase of the computation.

satisfiability of local constraint stores for goals that need to fail.

2. In a consistency phase, expand the constraint store in a way such that
   (a) it remains satisfiable and
   (b) other constraints local to the consistency phase become unsatisfiable.

Note that the first task is in fact the standard interface of the constraint solver in CLP. Note also that in the second task the constraint solver is used actively towards satisfying the top level goal by expanding the global constraint store.

2.3.1. Notations and Definitions

The following definitions formalise auxiliary notions exposed in the previous example, 2.2, needed for formalising the definition of the ACLP proof procedure.

We will first extend the language of the underlying CLP($\Re$) framework to contain a separate class of variables, called *naming variables or fixed variables*, obeying the same constraint theory as any other variable in CLP($\Re$). They will be denoted by n,m,..., possibly with a subscript.

Naming variables are used to name an existential variable shared in the abducible hypotheses Δ generated by the procedure and the goals of the computation.

Definition 2.12. (Naming abducibles)
An atom in a goal whose predicate is an abducible is called an abducible atom. Let A be an abducible atom with free variables $X_1, \ldots, X_j$. Then the naming of A, denoted by $nm(A)$, is the atom $A\theta$ where θ is a substitution $\{n_1/X_1, \ldots, n_j/X_j\}$ with $n_1, \ldots, n_j$ new naming variables. An abducible atom depending only on ground terms or naming variables is called a fixed abducible atom (or simply a fixed abducible).

Definition 2.13. (Negative constraints)
The negation, $\neg C$, of a set of constraints $C(X, n)$, depending on the free variables X and naming variables n, is given by $\neg \exists X C(X, n)$. Such a formula, $\neg C$, will be called a negative constraint.

Examples of negative constraints are:

$$
\begin{array}{ll}
C(X,n) = \{n = 2\} & \neg C(X,n) = \{\neg(n = 2)\} \\
C(X,n) = \{n = f(X), X = 3\} & \neg C(X,n) = \{\neg \exists X(n = f(X), X = 3)\} \\
C(X,n) = \{X = 2\} & \neg C(X,n) = \{\neg \exists X(X = 2)\}
\end{array}
$$

Definition 2.14. (Reverse Naming)
Let $A = \Delta \cup C \equiv \{a_1, \ldots, a_i\} \cup \{c_1, \ldots, c_j\}$, where Δ is a set of fixed abducibles, possibly containing naming variables, and C is a set containing positive or negative constraints. Let $n_1, \ldots, n_k$ be the naming variables occurring in A. Then the reversed naming of A, denoted by $Rn(A)$, is $\exists X_1 \ldots X_k(a_1^{'}, \ldots, a_i^{'}, c_1^{'}, \ldots, c_j^{'})$ where $X_1 \ldots X_k$ are new distinct variables and $a_s^{'}$ (resp.$c_s^{'}$) is obtained from a_s (resp.c_s) by replacing each occurrence of a naming variable n_t by a corresponding variable X_t. Similarly, we denote by $Rn(C)$ the conjunction of constraints

obtained from C by replacing the naming variables occuring in C by new existentially quantified variables.

Definition 2.15. (Solvability)
A set of constraints, C, is solvable iff $Rn(C)$ is solvable.

For example, in the domain of integers, if $C = \{n = 2\}$ then both C and $\neg C$ are separately solvable, since both $\exists Y(Y = 2)$ and $\exists Y \neg(Y = 2)$ are solvable, whereas if $C = \{X = 2\}$ then $\neg C$ is not solvable as $\neg\exists X(X = 2)$ is not solvable.

2.3.2. Abductive and Consistency Derivations

The proof procedure consists of two interleaving phases, called abductive and consistency phases, defined as follows.

Abductive Derivation

An abductive derivation in an ACLP theory $< P, A, IC >$ is a sequence

$$(G_1, \Delta_1, \Delta_1^*), \ldots, (G_k, \Delta_k, \Delta_k^*)$$

such that, for each $i \geq 1$, G_i is a goal of the form

$$\leftarrow C[]L_1, \ldots, L_s$$

C being a (possibly empty) set of constraints and $L_1, \ldots, L_s$ are atoms. Each Δ_i is a set of fixed abducibles and each Δ_i^* is a set of goals. An atom L_j is selected from G_i (G_i non-empty) and $(G_{i+1}, \Delta_{i+1}, \Delta_{i+1}^*)$ is derived according to one of the following rules :

(A1) if L_j is not abducible then
$G_{i+1} = S$
$\Delta_{i+1} = \Delta_i$
$\Delta_{i+1}^* = \Delta_i^*$
where S is the (CLP) resolvent [4]of some clause in P with G_i on L_j (and the new set of constraints C' in S is solvable).

(A2) if L_j is an atom with an abducible predicate and L_j resolves with a member of Δ_i returning the constraints C_1 and $C \cup C_1$ is solvable then
$G_{i+1} = \leftarrow C \cup C_1[]L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_s$
$\Delta_{i+1} = \Delta_i$
$\Delta_{i+1}^* = \Delta_i^*$

(A3) if L_j is an atom with an abducible predicate, let $nm(L_j) = L_j^{'}$ and $C'' = C \cup \{X = n\}$, where X is the set of variables in L_j and n is a set of corresponding new naming variables. Then, if there exists a consistency derivation for the abducible atom $L_j^{'}$, from
$(\{L_j^{'}\}, C'', \Delta_i \cup \{L_j^{'}\}, \Delta_i^*)$ to $(\{\}, C', \Delta', \Delta^{*'})$ then

[4] A CLP resolvent is an ordinary logic program resolvent from a clause and a goal with the additional requirement that the set of constraints in the resolvent are solvable.

$G_{i+1} = \leftarrow C'[]L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_s$
$\Delta_{i+1} = \Delta'$
$\Delta^*_{i+1} = \Delta^{*'}$

Step `(A1)` corresponds to an ordinary CLP derivation step. Rule `(A2)` reuses the already made abducible assumptions in the current set Δ_i. This step covers the following cases. A new existential assumption can be satisfied by reusing an older assumption. An old existential assumption may be specialised after unifying it with the new assumption. Finally, an old ground assumption can simply be reused. The aim of this rule is thus to allow for the possibility to reuse existing hypotheses and so to reduce the complexity of the explanation.

Step `(A3)` is concerned with generating new abductive hypotheses (when step `(A2)` cannot be applied or leads to failure). The new abducible atom is named before it is added to the current Δ. The constraint store is extended with the naming constraints which bind the variables in the abducible with their corresponding naming variables. The derivation continues with the remaining subgoal only if the new hypothesis is shown to be a consistent addition to Δ through a consistency derivation starting from it. In order to ensure this consistency the global constraint store and the set of goals Δ^* may need to grow in the consistency derivation. Note that under the restrictions we have imposed on the program P and integrity constraints IC of an ACLP theory the set of abducible hypotheses Δ does not grow inside a consistency derivation. For more general cases, as we will see below, it is also possible for this to grow in Δ.

The set of goals Δ^*, which appears in the definition of the abductive derivation, is generated in the consistency phases of the computation. As we will see below in the definition of the consistency derivation, the goals it contains are, in fact, exactly in the form of the restricted integrity constraints, i.e. denials with at least one abducible predicate appearing in their conditions, and they will be treated in exactly the same way. Hence they represent goals which must fail, if the overall computation is to be successful. The only difference they have with the integrity constraints in IC is that they are generated dynamically within the computation and thus are just local to the particular computation. In effect, these represent universally quantified assumptions on abducible predicates that need to hold for the goal to succeed.

Consistency Derivation

A consistency derivation in an ACLP theory $\langle P, A, IC\rangle$ for an abducible atom L, is a sequence

$$(L, C_1, \Delta_1, \Delta^*_1), (F_1, C_1, \Delta_1, \Delta^*_1), \ldots, (F_k, C_k, \Delta_k, \Delta^*_k)$$

where for each $i \geq 1$, C_i is a set of constraints, Δ_i is a set of fixed abducibles over the domain of $\Re$ extended with naming variables, Δ^*_i is a set of goals, and

1. F_1 is the union of all goals, no goal being empty[5], of the form $\leftarrow C_{di}[]L_1, \ldots, L_s$ obtained by resolving the abducible L with the denials in IC and Δ^*_1.
2. For each $i \geq 1$, F_i has the form $\{\leftarrow C_{di}[]L_1, \ldots, L_s\} \cup F'_i$, where F'_i is a set of

[5] An empty clause, or goal, is the clause $\leftarrow \{\}[]\{\}$, which has an empty constraint store and an empty body.

goals. The atom L_j or the local constraint store C_{di} (non-empty) is selected and $(F_{i+1}, C_{i+1}, \Delta_{i+1}, \Delta^*_{i+1})$ is obtained according to the following rules:

`(C1)` if L_j is not an abducible, let R be the set of all resolvents of clauses in P with $\leftarrow C_{di}[]L_1, \ldots, L_s$ on L_j. If no $r \in R$ is the empty clause, then
$F_{i+1} = R \cup F'_i$
$C_{i+1} = C_i$
$\Delta_{i+1} = \Delta_i$
$\Delta^*_{i+1} = \Delta^*_i$.

`(C2)` if L_j is an atom with an abducible predicate, let R be the set of all resolvents of $\leftarrow C_{di}[]L_1, \ldots, L_s$ with atoms in Δ_i on L_j. If no $r \in R$ is the empty clause, then
$F_{i+1} = R \cup F'_i$
$C_{i+1} = C_i$
$\Delta_{i+1} = \Delta_i$
$\Delta^*_{i+1} = \Delta^*_i \cup \{\leftarrow C_{di}[]L_1, \ldots, L_s\}$.

`(C3)` if C_{di} is selected then
$F_{i+1} = F'_i$
$C_{i+1} = C_i \cup C'$
$\Delta_{i+1} = \Delta_i$
$\Delta^*_{i+1} = \Delta^*_i$
where C' is a set of constraints (on the naming variables appearing in C_{di}) such that $C_i \cup C'$ is solvable and $C_{di} \cup (C_i \cup C')$ is not solvable.

A successful consistency derivation, as indicated in the rule `(A3)`, is one that ends with an empty set of goals F_k. In other words, this means that all goals in F_1 end in failure.

In a consistency derivation the current constraint store, C_i, is a set of hypotheses (like Δ_i) which is carried through the derivation and it defines the context in which the solvability of new constraints will be decided. Additional constraints may extend this constraint store through `(C3)`, in order to ensure the consistency of the integrity constraints, provided that the store always remains solvable.

Step `(C1)` splits the computation into as many branches as the number of resolvents of clauses in P with L_j. Note that resolvents which have constraints that are inconsistent with the current store C_i do not need to be considered, since these branches can never succeed in the context of C_i. The dropping of such a resolvent can be effected by a follow up trivial application of step `(C3)`, with the chosen set of constraints $C' = \{\}$.

Step `(C2)` deals with the failure of a branch due to an abducible condition L_j contained in the goal of this branch. If the computation has already made assumptions, in Δ_i, on the same abducible predicate as that of L_j then new branches are generated by resolving these old assumptions with L_j, as in step `(C1)`. Otherwise, this branch fails because of the abducible L_j, since the program contains no rules for the abducible predicates. In either case, the whole goal of this branch is remembered by adding it to Δ^*_i because it is possible during the incremental construction of the final Δ, at a later stage of the computation, to make new assumptions on the same abducible predicate which may unify with this goal on L_j. If particular search strategies are adopted in the computation then this may not be necessary. For example, if the consistency phases are delayed until all hypotheses on abducible predicates have been made, then there is no need to remember the goals in Δ^*.

In step (C3), the failure of the current branch is ensured (or forced) by extending the global constraint assumptions, C_i, with a new set of constraints C' which makes the local set C_{di} unsolvable. Thus this branch can be dropped. For example, one possibility is to have $C' = \neg C_{di}$ of the whole local constraint C_{di}. $\neg C_{di}$ would typically be equivalent to a disjunction of constraints. Note that C_{di} can first be simplified to an equivalent set of constraints by applying the naming constraints to replace local variables in C_{di} before we negate this. Also when negating this we can consider only a subset of its constraints, selecting C' to be the negation of the constraints on some naming variable. Typically, it is more useful to negate parts of the constraint that refer to naming variables as this will constrain further the abducible assumptions which involve these variables. For example, if we have $C_{di} = \{Y = n, Y > 4, Z < 7\}$ this can be simplified to $C_{di} = \{n > 4, Z < 7\}$ and the negating constraint C' can be choosen to be $n \leq 4$. Note also that, if in any of the other steps of the consistency derivation the local constraint store C_{di} of the resolvent goal is not solvable together with the current global constraint store C_i, then an (immediate) application of step (C3) (with $C' = \{\}$) allows us to drop this branch.

The above proof procedure is sound with respect to the semantics for ACLP defined in section 2.2.

Definition 2.16. Let $\langle P, A, IC\rangle$ be an ACLP theory. A successful derivation is an abductive derivation that ends with a triple of the form $(\leftarrow C[]\{\}., \Delta, \Delta^*)$.

Theorem 2.3. Let $\langle P, A, IC\rangle$ be an ACLP theory where P is a definite program and the integrity constraints in IC are of restricted form (as in definition 2.11). Let also the sequence $(\leftarrow G, \{\}, \{\}), \ldots, (\leftarrow C[]\{\}\ ., \Delta, \Delta^*)$ be a finite (successful) abductive derivation. Define $\Delta_{rn} = Rn(\Delta \cup \exists XC)$, where X is the set of free variables (not naming variables) appearing in the resulting constraint store C. Then Δ_{rn} is an abductive explanation for the goal G.

PROOF. The proof of this theorem is based on a mapping of the abductive and consistency derivations, that the successful abductive derivation contains, into successful and (finitely) failed CLP derivations respectively, for the program $P \cup \Delta_g$, where Δ_g is any ground (over $D\Re$) set of abducibles such that $\Delta_g \models_{\Re} \Delta_{rn}$, and then using the theorems for the soundness of CLP derivations. The full proof is given in the appendix.

We note that the constraint store C, computed by the successful derivation, constitutes an answer constraint for the goal G. It contains equality constraints that link the naming variables in the abducibles of Δ with a subset of the variables in the goal G. A grounding valuation g of the computed constraint store C that grounds the abducibles in Δ to Δ_g will give a corresponding grounding, G_g, of the goal G, for which Δ_g is a ground abductive explanation.

The ACLP proof procedure is incomplete. This procedure is based on Eshghi and Kowalski's [14] and Kakas and Mancarella's [25] proof procedures which are also resolution (rewriting) based and suffer from incompleteness. One of the main reasons for their incompleteness is that they cannot deal with infinite derivations resulting from positive loops in the definition of predicates in the program P (eg. from $p \leftarrow p$). Other procedures, such as the IFF proof procedure [15] and SLDNFA [7], use a weak form of semantics like that of the three-valued completion to obtain completeness results. The stronger minimal-model semantics of ACLP makes it more difficult to obtain completeness. For this it will be necessary to deal with infinite derivations in an analogous way to that studied by Przymusinski in [47] with the SLS procedure, where effectively an infinite derivation in the consistency phase is considered to be a derivation that leads to failure.

2.4. Extensions of the basic ACLP framework

In this section we present the extension of the basic framework of ACLP to allow negation as failure (NAF) in the program P and the denials of the restricted integrity constraints IC of an ACLP theory $\langle P, A, IC \rangle$. We will also discuss how the proof procedure for the computation of ACLP can be used with more general forms of integrity constraints by appropriately transforming these constraints into the restricted denial form that can be handled directly by the ACLP proof procedure extended to include also NAF.

The general form of ACLP theories containing NAF that we will consider are defined as follows.

Definition 2.17. (General ACLP theory)
A general ACLP theory is an ACLP theory $\langle P, A, IC \rangle$ as given in definition 2.4 where the constraint logic program P can contain negation as failure conditions (on non-abducible predicates) in the body of its rules. The integrity constraints IC are of the restricted form given by definition 2.11 but where some of the conditions in the denial can be negation as failure literals on non-abducible predicates[6].

One way to extend ACLP to allow NAF is to first lift the semantics of NAF for CLP (see e.g [40], [22]) and then apply analogously the abductive semantics as in the previous section. Another way which is particularly appropriate for the ACLP framework is to use the *abductive* interpretation of NAF proposed in [14] and further studied in [24, 9]. In this treatment of NAF negative literals are interpreted as abductive hypotheses that can be assumed to hold provided that, together with the program, they satisfy a canonical set of integrity constraints.

Using this abductive interpretation of NAF the semantics of a general ACLP theory can be identified with that of a corresponding (positive) ACLP theory where no NAF appears of the form studied in the previous section. This correspondence (based on that of [14, 24]) is given as follows.

Definition 2.18. (Positive version of an ACLP theory)

[6] We will see below that the semantics of the integrity constraints as true formulae in the computed model of the theory means that negative conditions in denial integrity constraints can be interpreted equivalently either as NAF (as in this definition) or as classical negation.

Let $T = \langle P, A, IC \rangle$ be a general ACLP theory. The corresponding positive ACLP theory $T^* = \langle P^*, A \cup A^*, IC^* \cup I^* \rangle$ of T is given as follows:

- A new predicate symbol p^* (the opposite of p) is introduced for each non-abducible p in P. The set of abducibles A^* is the set of all these predicates.
- P^* is P where each negative literal $not\, p(t)$ in the rules of P has been replaced by $p^*(t)$.
- IC^* is IC where each negative literal $not\, p(t)$ in the denials of IC has been replaced by $p^*(t)$.
- I^* is the set of all, for each non abducible predicate p, integrity constraints of the form:

$$\forall X \neg [p(X) \wedge p^*(X)] \text{ and } \forall X [p(X) \vee p^*(X)].$$

Similarly, we tranform any goal G to a new goal G^* by replacing each negative literal *not* $p(t)$ in G by $p^*(t)$.

Under this transformation the semantics of a general theory T is given by the semantics of its transformed positive version T^* as defined above in section 2.2. An abductive explanation will now contain two types of abducible assumptions: assumptions on the original incompletely defined predicates in A and NAF assumptions on predicates from A^*. Otherwise, the central definition of an abductive explanation, that gives the semantics of T^* and hence of T, remains the same as above in definitions 2.8 and 2.9.

As shown in [14] (and [24]) this abductive treatment of NAF in logic programs corresponds to the Stable Model [17] (and Generalized Stable) model semantics. Other semantics such as the partial stable model or the well-founded model semantics can also be accommodated within the abductive treatment of NAF by dropping the disjunctive integrity constraint for the NAF abducibles from I^* and adopting an argumentation framework for the acceptance of the NAF abducibles (now treated as arguments). This requires that instead of the disjunctive constraint we impose an extra argumentation-based condition on any set of NAF abducibles (arguments) Δ that we want to allow as part of an abductive explanation. This condition states that whenever for some (other) set of NAF abducibles A, we have $P^* \cup A \models p$ with $not\, p \in \Delta$ (we then say that A attacks Δ) then it must also be the case that $P^* \cup \Delta \models q$ with $not\, q \in A$, i.e. Δ must also attack A. The full exposition of this argumentation theory is beyond the scope of this paper and the reader is referred to [23] for a survey of this.

For simplicity, we will assume that we are considering a class of logic programs in our general ACLP theories, such as that of order-consistent [49] programs, for which the use of the disjunctive constraints or the extra argumentation-based condition is equivalent [7]. In fact, below when we consider the extension of the ACLP proof procedure to include NAF it will not be necessary to have the disjunctive constraints explicitly in the integrity constraints of the ACLP theory with which we are carrying

[7] For such cases the stable model semantics always exists and coincides with the partial stable model semantics.

out the computation. Their satisfaction will be quarranteed by a suitable extension of the proof procedure in line with their equivalent extra argumentation condition.

In our definition of a general ACLP theory we have assumed that NAF will be used only on non-abducible predicates as it would not be appropriate to apply this type of negation on the abducible predicates which by definition are not completely defined in an ACLP theory. We note here that negation on abducibles can be treated again in a simple way through abduction by introducing another different class of abducible predicates symmetrically to the original abducible predicates in A to represent negative assumptions on them. Therefore for each abducible predicate a in A we have also the abducible predicate $\overline{a}$ for the negation of a together with the additional integrity constraint $\forall X \neg [a(X) \wedge \overline{a}(X)]$ but not the disjunctive constraint as for NAF. Again all definitions hold as before in the new theory where any negation on an abducible a in the program P is replaced by the corresponding positive atom on $\overline{a}$ and the above integrity constraints are added to the theory.

Together with the abductive interpretation of NAF the authors in [14] have given an abductive proof procedure for its computation. This was shown in [9] to be correct under the argumentation based semantics of preferred models or its equivalent [28] partial stable model semantics. The proof procedure for ACLP given above is in fact based on this procedure and its extensions studied in [25, 27]. It can therefore be naturally extended to allow the computation of ACLP to include NAF as an extra type of abducible. The abductive derivation of the procedure remains essentially the same, treating the NAF abducibles in the same way as any other abducible. When a NAF abducible q^* is encountered and added to the computed Δ its consistency checking will lead, via the integrity constraint $\leftarrow q^*, q$, to a consistency derivation to ensure the failure of the goal $\leftarrow q$.

A new element in the definition of the consistency derivation is the fact that now, in order to fail on a NAF abducible[8] p^*, it is necessary to start a new subsidiary abductive derivation rooted at the goal $\leftarrow C[]p$ where C is the current computed constraint store. This attempt to show p can then require in turn the addition of further abductive assumptions to the set of hypotheses Δ which is being generated. If indeed we can show p possibly by adding new abducible assumptions to Δ then the assumption p^* can not be supported and the consistency goal that contains it fails as required for the satisfaction of its parent integrity constraint. This new step in the consistency derivation is directly related to the argumentation interpretation of NAF where p^* belongs to an attack against Δ and the proof of p by the subsidiary abductive derivation ensures that this attack is attacked back by Δ.

We note again that both the global constraint store C and the set of abductive hypothese Δ can grow in a consistency phase giving a full interleaving between abductive and consistency phases of the computation.

The formal extension of the ACLP proof procedure as defined above, needed to accommodate NAF is given by the addition of the following extra step, $C4_{naf}$, in the definition of the consistency derivation. Goals are computed by this extended procedure in their positive form using the positive form of the given general ACLP theory.

[8] A NAF condition may appear in a consistency derivation goal from an integrity constraint or after a previous resolution step with a rule in the program.

$(C4_{naf})$ if L_j is a ground or fixed NAF literal [9]of the form $p^*(n)$ (where n is the set of naming variables appearing in L_j after the naming constraints in the local constraint store C_{di} have been applied), $p^*(n) \notin \Delta_i$, the set R of all resolvents of $\leftarrow C_{di}[]L_1, \ldots, L_s$ with atoms in Δ_i on L_j does not contain the empty clause and there exists a finite successful abductive derivation from $(\leftarrow C_i[]p(n), \Delta_i, \Delta_i^*)$ to $(\leftarrow C'[]\{\}, \Delta', \Delta^{*'})$, then
$F_{i+1} = R \cup F_i'$
$C_{i+1} = C'$
$\Delta_{i+1} = \Delta'$
$\Delta_{i+1}^* = \Delta^{*'}$.

Note that in the subsidiary abductive derivation to prove $p(n)$ we carry along the (global) constraint store C_i,the set of assumptions Δ_i and the reduced integrity goals Δ_i^*) computed so far. These can then grow within this subsidiary derivation.
The extended ACLP proof procedure remains sound.

Theorem 2.4. Let $T = \langle P, A, IC\rangle$ be a general ACLP theory where P is an order-consistent program. Let also the sequence $(\leftarrow G^*, \{\}, \{\}), \ldots, (\leftarrow C[]\{\} ., \Delta, \Delta^*)$ be a finite (successful) abductive derivation. Then $\Delta_{rn} = Rn(\Delta \cup \exists XC)$, where X is the set of free variables (not naming variables) appearing in the resulting constraint store C is an abductive explanation for the goal G^* in the positive form $T^* = \langle P^*, A \cup A^*, IC^* \cup I^*\rangle$ of the theory T.

Proof. The proof of this theorem is given in the appendix. It uses similar methods as the proof of soundness of the simpler ACLP procedure above by mapping the successful abductive derivation into a successful derivation of the Eshghi and Kowalski procedure in [14] for ground abduction of NAF (on which the ACLP proof procedure is based) and appealing to the soundness of this procedure as shown first in [9].

Note that this result shows that the extended ACLP proof procedure is sound under the stable model (or generalized stable model) semantics for NAF. In fact, as mentioned above by droping the disjunctive constraint in I^*, which in anycase does not play any direct role in the computation of the extended ACLP procedure, the soundness result of [9] for the Eshghi and Kowalski procedure allows us to show that the extended ACLP procedure is also correct with respect to the partial stable models semantics for NAF irrespective if the program P of the given general ACLP theory is order-consistent or not.

We also note that the treatment of NAF in a consistency phase is limited in that the procedure is allowed to select only fixed NAF abducibles whose variables are determined, i.e. where each variable takes the value of some ground term or of some naming variable, by the current collection of global constraints and the local constraints in the goal from which the NAF abducible is selected. Otherwise, the procedure flounders. Extending the procedure to select non-fixed NAF abducibles in a consistency phase will essentially require the adoption of a universally quantified assumption together with an appropriate extension of the consistency check for such hypotheses. Again this is beyond the scope of this paper.

[9]A fixed NAF literal (like a fixed abducible) is a NAF literal depending only on ground terms or naming variables.

Although the semantics of ACLP allows any formula as an intregrity constraint its proof procedures as given above require that these have a special denial form. This allows the ACLP procedure to be in both its abductive and consistency phases an essentially backward reasoning goal-oriented process. More general forms of integrity constraints can be accommodate by exploiting the fact that their meaning is formalised within the ACLP semantics in terms of their truth value in a classical two valued computed model. This means that any classical truth preserving transformation on an integrity constraint does not change its meaning. We can therefore transform other forms of integrity constraints into denial form by such equivalence transformations.

In transforming the integrity constraints we need to relate the classical negation that would appear in the conditions of a denial to NAF conditions. The following result holds.

Proposition 2.1. Let $T = \langle P, A, IC\rangle$ be a general ACLP theory where P is an order-consistent program and $T^* = \langle P^*, A \cup A^*, IC^* \cup I^*\rangle$ its positive form. Let also ϕ be a (classical) denial of the form $\neg(B_1 \wedge \ldots \wedge B_m \wedge \neg A_1 \wedge \ldots \wedge \neg A_k)$ and $\overline{\phi}$ be the denial $\leftarrow (B_1, \ldots, B_m, notA_1, \ldots, notA_k)$. Then for any set of ground abducibles Δ_g that satisfies I^* (i.e. $M^{\Re}_{P^* \cup \Delta_g} \models I^*$), $M^{\Re}_{P^* \cup \Delta_g} \models \phi$ iff $M^{\Re}_{P^* \cup \Delta_g} \models \overline{\phi}$.

Proof. The formula ϕ is satisfied in a computed model $M^{\Re}_{P^* \cup \Delta_g}$ iff $M^{\Re}_{P^* \cup \Delta_g} \not\models B_j$ for some $j = 1, \ldots, m$ or $M^{\Re}_{P^* \cup \Delta_g} \not\models \neg A_i$ for some $j = 1, \ldots, k$. In the first case clearly this means that the denial $\overline{\phi}$ is also satisfied and vice-versa. In the second case, this means that $M^{\Re}_{P^* \cup \Delta_g} \models A_i$ which implies that $notA_k \notin \Delta_g$ as otherwise I^* would not be satisfied. Hence $M^{\Re}_{P^* \cup \Delta_g} \not\models notA_i$ and $\overline{\phi}$ is satisfied.
Conversely, if $\overline{\phi}$ is satisfied because $notA_k \notin \Delta_g$ then the satisfaction of the disjunctive constraints in I^* implies that $M^{\Re}_{P^* \cup \Delta_g} \models A_i$ and hence ϕ is satisfied by $P^* \cup \Delta_g$.

Hence under the stable (and generalized stable) model semantics for NAF the two forms of negation in integrity constraints of denial form are equivalent. For an order-consistent program P this semantics always exists and coincides with that of the partial stable models. If we drop the order-consistent requirement and adopt the partial stable model semantics for NAF it is possible for $notA_i$ to be false without A_i to hold and hence this equivalence does not hold. Nevertheless, it is important to note that for the puproses of computation we can still replace the classical negation conditions with NAF conditions since the extended ACLP proof procedure will satisfy a denial on a $notA_i$ condition only if it manages to prove A_i in a subsidiary abductive derivation for this. Hence the extended proof procedure remains sound under this replacement.

An important class of integrity constraints that can be transformed into denial form is that of closed sentences that can be logically transformed into clausal form without the introduction of skolem constants.

Proposition 2.2. Let $T = \langle P, A, IC\rangle$ be a general ACLP theory where P is an order-consistent program. Let also ϕ be an integrity constraint of the clausal form $A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_m$ and $\overline{\phi}$ be the denial $\leftarrow (B_1, \ldots, B_m, notA_1, \ldots, notA_k)$. Then ϕ is sastisfied iff $\overline{\phi}$ is sastisfied.

PROOF. The result is a direct consequence of the proposition above by transforming ϕ into its logically equivalent form $\phi' = \neg(B_1 \wedge \ldots \wedge B_m \wedge \neg A_1 \wedge \ldots \wedge \neg A_k)$.

When we transform a general integrity constraint into the denial form given by $\leftarrow (B_1, \ldots, B_m, notA_1, \ldots, notA_k)$, if none of the $B_1, \ldots, B_m$ are abducible we can then unfold one (or more) of these positive conjuncts with each of the rules in the program P^* of our ACLP theory for the predicate of this conjunct. If $\{\beta_1, \ldots, \beta_n\}$ is the set of all the bodies of the clauses whose heads unify with some selected atom, say B_1, with $\{\theta_1, \ldots, \theta_n\}$ as corresponding mgus, then this unfolding will replace the denial above by the set of constraints:$(j = 1, \ldots, n)$, $\leftarrow (\beta_j, \ldots, B_m, notA_1, \ldots, notA_k)\theta_j$. It is easy to see that the original denial is satisfied iff all of the unfolded denials are satisfied. This unfolding is repeated until an abducible appears in each of the generated denials. In some cases, this simple unfolding process may not terminate due to recursive predicates. More advanced forms of unfolding that employ loop checking can be developed to help overcome this problem for particular cases of non-termination.

For other forms of integrity constraints, that cannot be brought into the desired denial form by classical equivalence transformations we can apply transfomations analogous to the Lloyd Topor transformations [39] using new auxiliary predicates in the program of our theory. These constraints would be transformed to define a new (arity zero) predicate $false$ and we would add at the end of any goal the extra condition of $not\ false$ i.e. of $false^*$. We should note though that this can often lead to floundering on NAF conditions in the consistency phase as described above e.g. when the assumption required for the satisfaction of the integrity constraint that has been transformed is essentially a universally quantified one such as $\forall X a(X)$. In this case a full first order computational method would need to be employed.

3. APPLICATIONS IN ACLP

The integrated framework of ACLP can be used to formalize and solve a variety of applications. Abduction and Abductive Logic Programming has been successful in formalizing many problems all of which can also be addressed within ACLP. Applications within ACLP are based on the view that abduction is a process that generates solutions in terms of the problem specific abducibles. For each problem we define its own answer holders, the abducible predicates. This property, of having an explicit problem specific answer holder, helps in developing high-level and flexible representations of the problems, as we can now specify more directly properties of the solution and, in general, the basic structure of the problem can be modelled in a more direct and faithful way. From the point of view of Constraint Logic Programming the framework of ACLP offers an environment where we can express our constraints more directly than in a CLP framework alone. Constraint solving in CLP is exploited at a lower-level to where the high-level constraints are reduced via abductive reasoning.

This section presents an implementation of the ACLP framework, based on the proof procedure developed in the previous section, and its application to the two problems of planning and job-shop scheduling. Other areas of applications, where ACLP can be used, includes (temporal) diagnosis, updating temporal databases and in general any abduction problem with a rich temporal and/or spatial structure. A simple form of the ACLP framework has also been successfully applied to two real-life problems: a university course timetabling [31] and an airline crew-scheduling problem, dealing both with the problem of finding an initial solution and the problem of rescheduling under the face of new information. Details on these later applications can be found in [43].

Through the study of these application problems we aim (i) to illustrate the co-operation of abduction and constraint solving within the computation of ACLP and the enhancing effect that this has on abduction, and (ii) to test the computational effectiveness and flexibility of the ACLP framework.

Two types of experiments, on the applications of planning and job-shop scheduling, are performed with the ACLP system and presented here. The first type tests the computational viability of the ACLP system and its performance, in comparison with the direct use of the (lower level) constraint solving framework of CLP on which ACLP is built. The other type of experiments aims to illustrate the high level expressivity of ACLP and the flexibility it can provide in adapting a program to new requirements or changes in the problem definition.

3.1. Implementation of ACLP

The ACLP system has been implemented on top of the ECLiPSe language as a meta-interpreter, using explicitly the low-level constraint solver that handles constraints over finite domains (integer and atomic elements). The naming variables of ACLP are implemented through domain variables of ECLiPSe. A difficult task of the meta-interpreter is the management of the interleaving phases of abduction and consistency during a computation in the ACLP framework. Since constraints can be imposed on the (naming) domain variables, not only during the generation of an abducible assumption but also during consistency checking, there is a need for: (i) dynamic generation and management of low-level CLP constraints and (ii) management of the domain variables. The implementation contains a module for negating CLP constraints and sending them to the ECLiPSe constraint solver to be satisfied together with the other constraints already set. A binding environment is constructed during the execution of the meta-interpreter, where the domain variable bindings and the constraints attached to these variables are stored. An explicit mechanism is implemented to handle the attachement of new constraints to variables and the linking between different constraints that are imposed on the same domain variable at different parts of the computation.

In the ACLP system a program is a finite set of ECLiPSe clauses together with a set of declarations of abducibles predicates in the form of ECLiPSe facts as: $abducible\_predicate(predicate\_name/arity)$, and a set of integrity constraints written as ECLiPSe rules of the form:

$$ic :- B_1, \ldots, B_n. \quad n \geq 1.$$

where:

- at least one of the goals $B_1, \ldots, B_n$ has an abducible predicate and
- the rest of the goals can be either user-defined predicates or constraint predicates of ECLiPSe.

The constraint predicates that can be used in the body of a program rule or an integrity constraint can be (i) *arithmetic constraint predicates (over the integers)* or (ii) *logical contraint predicates*. The arithmetic constraint predicates that are supported by the current ACLP implementation are:

`T1##T2` : the value of the term T1 is not equal to the value of the term T2.

`T1#=T2` : the value of the term T1 is equal to the value of the term T2.

`T1#<T2` : the value of the term T1 is less than the value of the term T2.

`T1#<=T2`: the value of the term T1 is less than or equal to the value of the term T2.

`T1#>T2` : the value of the term T1 is greater than the value of the term T2.

`T1#>=T2`: the value of the term T1 is greater than or equal to the value of the term T2.

The system also supports logical constraints such as conjunction, #$\wedge$, and disjunction, #$\vee$. These simple constraints can be combined to build complex logical constraint expressions.

During a consistency phase for the consistent addition of a new abducible assumption, `a(X)` where X is now a "marked" ECLiPSe domain variable, constraints are negated and set in the current constraint store. This negation of the constraints is the usual mathematical negation, e.g. the negation of the aritmetic constraint `T1#<T2` is `T1#>=T2`. The negation of the logical constraint #$\wedge$ is #$\vee$. For example, the negation of the logical constraint `T1##5 #`$\wedge$` T2 #>2` is `T1#=5 #`$\vee$` T2 #<=2`.

In the current implementation of ACLP, a program is executed by calling the top-level ECLiPSe query:

aclp-solve(goal, initial-hypothesis, output-variable)

The output variable returns a list of abducible hypothesis, with their domain variables constrained according to the dynamic constraints that were generated through the unfolding of the "relevant" part of the program and the integrity constraints. A subsequent step of *labeling* on these variables gives a ground solution of our initial query. Various other constraint predicates of ECLiPSe can be used at this stage e.g. *min_max/2* or *minimize/2* in order to find an optimal ground solution.

For clarity of presentation, in the rest of this section we will continue to use the notations of the previous section instead of the specific programming notation of the ACLP system.

3.2. Abductive Planning in ACLP

We now present the problem of abductive planning in ACLP in order to illustrate through this example the main features of the framework when this is applied to an application problem. We will give particular attention to the computation of ACLP and its search together with the role of the constraint solver in this.

Several authors [13], [33], [53] have studied planning as an abductive problem. Indeed, this can easily be formalised through abduction in *ACLP* as follows. Suppose that we have a general theory of reasoning about actions, we then consider an ACLP theory $< P, A, IC >$ where:

- the program P contains this general theory of actions together with any specific axioms of the effects of actions for the particular planning domain of our application.
- the abducible predicates in A are the operator (action) predicates in the particular domain of our planning problem
- the integrity constraints IC encode the precondition requirements that must hold for each type of possible action.

For example, if we use the Event Calculus [36] as our basic theory for reasoning about actions abductive planning is formalized within the framework of *Abductive Event Calculus* as follows. The program P of the abductive theory $\langle P, A, IC\rangle$ contains the basic axioms[10]:

$$\begin{aligned} holds(F,\, T_2) \leftarrow\ & happens(E, T_1),\\ & T_1 \leq T_2,\\ & initiates(E,\, F),\\ & not\ clipped(T_1,\, F,\, T_2) \end{aligned}$$

$$\begin{aligned} clipped(T_1,\, F,\, T_2) \leftarrow\ & happens(E, T),\\ & T_1 < T < T_2\\ & terminates(E,\, F) \end{aligned}$$

These general axioms express the persistence of a property F from the time T_1 at which it is initiated by an event (i.e. a specific action) E to a later time T_2. This property can persist provided that it is not *clipped* by another event (specific action) that has the effect of terminating such properties. The general effects of actions initiating or terminating properties are given by other domain specific axioms in the program P defining the $initiates(E, F)$ and $terminates(E, F)$ predicates.

The predicate *happens* is not defined in P and is designated as an abducible predicate. Hence $A = \{happens\}$. A plan will consist of a set of assumptions on this abducible predicate.

The integrity constraints IC contain statements of the form

$$holds(F, T) \leftarrow happens(E, T), precondition(E, F)$$

expressing the fact that we can only assume that an event E has happened at time T if all its preconditions F hold true at this time. The predicate *precondition* is defined within the program P. Other more particular constraints on the possible actions of our domain will also be formalized as integrity constraints in IC.

A particular planning problem in the domain described by an abductive theory $\langle P, A, IC\rangle$ can then be solved by adding into the program P a description, S_i, of the particular initial state of our problem and setting the required final state, S_f, as an abductive goal. An abductive solution Δ to this goal, as specified in the previous section, will be an abductive explanation of S_f consisting of a set of actions and constraints with specific ground instances, Δ_g, satisfying the following conditions:

- $P \cup S_i \cup \Delta_g \models_{\Re} S_f$, and

[10] For clarity of presentation we will not separate the domain constraints (equality or inequality) at the front of the body of a program rule or an integrity constraint but leave these at their "natural" position.

- $P \cup S_i \cup \Delta_g \models_{\Re} IC$

where $\Re$ is the constraint structure for the lower-level constraints of equality, inequality and other more specific constraints in our particular planning domain. Here we will assume that $\Re$ is the usual arithmetic domain for the integers.

The first condition implies that when these actions are performed the initial state of the problem will be transformed into the final required state. The second condition ensures that the assumed actions can indeed be performed i.e. their preconditions (will) hold at their time of execution. In general, the assumed actions, in the abductive solutions for planning problems, have a temporal dimension and often also a spatial dimension, as we will see below. This makes the integration to constraint solving particularly interesting and beneficial.

3.2.1. Planning in the Blocks World

Let us consider the particular simple but instructive planning domain of the blocks world. Its abductive formalization, based on the general formulation above of the abductive event calculus, is as follows (here we have simplified the presentation by specializing the above general theory for the simple domain at hand stripping also away the *holds* and *happens* predicates). The abductive theory $\langle P, A, IC\rangle$ has the following general axioms in P:

$$
\begin{array}{ll}
on(X,Y,T) & \leftarrow initially(X,Y,T_0), T_0 \leq T, not\ moved(X,T_0,T). \\
on(X,Y,T) & \leftarrow move(X,Y,T_1), T_1 \leq T, not\ moved(X,T_1,T). \\
moved(X,T_1,T_2) & \leftarrow move(X,Y,T_3), T_1 < T_3 < T_2, Y \neq X.
\end{array}
$$

where $on(X,Y,T)$ is a (the only) property predicate, meaning that block X is on block or position Y at time T, and $initially(X,Y,T_0)$ means that block X is at position Y at an initial time T_0. The initial state of the blocks is then described by a collection of facts of these predicates. The predicate $moved(X,T1,T2)$, corresponding to the *clipped* predicate, is defined to hold if block X was moved within the time period T_1 to T_2 thus terminating the initiated position of X at T_1. Note that in this simple formulation of the problem we are assuming that the effects of moves start to hold instantaneously.

The abducible predicates are $A = \{move(-,-,-)\}$ as we only have one type of action in the example domain that we are considering. This action of $move(X,Y,T)$ means that block X is moved to position Y (that may be another block) at time T. The general integrity constraints for the preconditions of this action are the two constraints which state that a block X can be moved to a new position Y only if the block X and the position Y are both clear i.e. there is no block on top of either of them. These can be represented by the following two denials in IC:

$$\leftarrow move(X,Y,T), on(Z,X_1,T), X = X_1.$$

$$\leftarrow move(X,Y,T), on(Z,Y_1,T), Y = Y_1.$$

Note how these integrity constraints include explicit equality in them. The equality predicate is one of the constraint predicates in ACLP which can be handled explicitly by this framework. In ACLP the satisfaction of an integrity constraint, like the ones above, can be achieved by adding in the global constraint store (via step (C3) of the proof procedure) the negation of such equality (or other) constraint that appears in the integrity constraint.

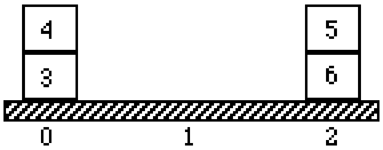

FIGURE 3.1. An example initial state

Note also that negation as failure is treated through abduction (see previous section and [14]) and hence all occurances of *not* $moved(-,-,-)$ in the program P will be replaced by a new predicate $moved^*(-,-,-)$. Furthermore, the set of integrity constraints IC will be extended with the additional constraint:

$$\leftarrow moved^*(X,Y,T), moved(X,Y,T).$$

Given the above formulation and an initial configuration (state) of the blocks described by a set of atomic facts on the *initially* predicate the aim is to find an abductive solution in terms of assumptions on the predicate *move* for a new (again given) configuration of the blocks. The moves in such a solution would constitute a valid plan that would allow us to reach the new configuration of the blocks from the initial configuration. We will assume that we are interested in linear plans where only one move can occur at any single time point.

Consider for example the initial configuration as shown in figure 3.1 represented by the following facts in P:

$initially(3,0,0)$. $initially(4,3,0)$. $initially(6,2,0)$. $initially(5,6,0)$.

and suppose that the new state requires that block 3 must be on block 6 (with the rest of the blocks free to be at any position). Thus our abductive goal will be to show $on(3,6,T_f), 0 < T_f$ where T_f is some (final) time at which the new state will hold. One abductive ACLP solution to this goal is given by:

$\Delta = \{move(3,6,t_1), move(4,5,t_2), move(5,1,t_3)\}$ and $C = \{t_2 < t_1, t_3 < t_1, t_3 < t_2\}$, where t_1, t_2, t_3 are naming variables that denote respective times for the three moves. Note that in fact the set Δ of abducible assumptions will also contain the NAF assumption $\{moved^*(3,t_1,t_f)\}$ to ensure that block 3 stays on 6 after it is moved there.

The high-level representation of the problem in $ACLP$ allows us to accommodate in a simple and direct way other more specific requirements that a particular case of the problem may require. These are often included as additional integrity constraints in IC. In particular, if (as we say above) we are interested in linear plans we can impose this by the extra integrity constraints

$$\leftarrow move(X_1,Y_1,T_1), move(X_2,Y_2,T_2), X_1 \neq X_2, T_1 = T_2.$$

$$\leftarrow move(X_1,Y_1,T_1), move(X_2,Y_2,T_2), Y_1 \neq Y_2, T_1 = T_2.$$

As another example, if it is required that some specific position c_y can not be used as a destination for any move during some (open) time period from c_1 to c_2 then we would add the intrgrity constraint

$$\leftarrow move(X,Y,T), T > c_1, T < c_2, Y = c_y.$$

Similarly, if when we move any block then each one of a group of related blocks must be at some given positions we would add the constraint

$$on(X_1, Y_1, T) \leftarrow move(X, Y, T), related(X, X_1), position(X_1, Y_1).$$

More examples of such extra constraints are given in the experiments section below.

3.2.2. Abductive Computation of Plans - Example

We will now analyze in some detail the abductive computation in ACLP (as defined by the proof procedure of the previous section) for the example problem above, in order to highlight some of the main features of the abductive computation with particular emphasis on the role of the constraint solver.

At first an abductive phase rooted at the required goal will reduce (step `(A1)` of the ACLP proof procedure) $on(3,6,T_f)$ and generate (step `(A3)`) the assumption $\Delta_0 = \{move(3,6,t_1)\}$, where t_1 is a new "naming" variable treated as a CLP domain variable. Δ_0 will also contain (through step `(A3)` again) a NAF assumption $moved^*(3, t_1, t_f)$. The treatment of these NAF assumptions is the same as other assumptions. We will discuss its consisistency checking at the end of this section.

A constraint store will also be formed with $C_0 = \{0 < t_1 \leq t_f\}$. Once this assumption is generated a consistency phase starts to ensure that the integrity constraints remain satisfied. This will reduce (step 1 of a consistency derivation) the two integrity constraints in IC to the following two goals[11]

$$\leftarrow on(Z, X_1, t_1), 3 = X_1 \qquad\qquad \leftarrow on(Z, Y_1, t_1), 6 = Y_1$$

requiring that both of these fail. Using the definitions of the predicates *on* and *initially* the first of the above reduced constraints expand further (step `(C1)`) to the following two branches where again both of these goals must fail:

$$\leftarrow 0 \leq t_1, moved^*(4, 0, t_1)$$

$$\leftarrow T_1 \leq t_1, move(Z, X_1, T_1), moved^*(Z, T_1, t_1), 3 = X_1$$

At this stage of the computation the second goal fails trivially (with no new additional constraints) as there is no other assumption of a move in the current set of hypotheses Δ_0.

Note here that when we have a move assumption (eg. move(b,P,t)) in Δ, then this second goal can be satisfied by adding in the global constraint store the constraint $3 \neq P$ coming from the explicit equality constraint $3 \neq X$ in the reduced integrity constraint. The first of these goals can fail, within an ACLP consistency phase, in two ways. One way is to add (step `(C3)`) the negation of $0 \leq t_1$ (ie. $t_1 < 0$) in the constraint store C_0. This though will not be allowed since it will make the constraint store inconsistent. The other way is to fail on $moved^*(4, 0, t_1)$ (step $(C4_{naf})$). This will result in a new subsidiary abductive phase to try to prove the goal

$$\leftarrow moved(4, 0, t_1).$$

This shows a simple example of a general strategy of the ACLP computation where

[11] Again for clarity of presentation, we will not show the unification equality constraints explicitly when these do not involve naming variables but instead apply these to the rest of the goal.

it first tries to ensure the satisfaction of the integrity constraints by generating additional arithmetic CLP constraints. Failing this the abductive computation will attempt to generate a new hypothesis (a new move).

Using the definition of the predicate *moved* this goal is reduced to (step `(A1)`)

$$\leftarrow move(4, Y, T), 0 < T, T < t_1, Y \neq 4.$$

The assumption set Δ_0 will then grow (step `(A3)`) to $\Delta_1 = \{move(3, 6, t_1), move(4, y_2, t_2)\}$, where y_2, t_2 are new "naming" variables - CLP domain variables. The domain of the variable y_2 is set to be equal to $\{0, 1, 2 \ldots, 6\}$ namely equal to all the available positions in our example problem. The constraint store will now contain $C_1 = \{t_1 > 0, 0 < t_2, t_2 < t_1, y_2 \neq 4\}$. Hence the consistency checking of one abducible, $move(3, 6, t_1)$, requires the generation of further assumptions. This new abductive hypothesis assumes that block 4 is to be moved at some location y_2 as yet unspecified, (apart from the fact that it must be different than 4) at some time t_2 also only partially specified by the requirement that it is before t_1.

A subsidiary consistency phase will check the consistency of the new abducible $move(4, y_2, t_2)$, by checking that the integrity constraints in IC remain satisfied. This will generate, as above, through step 1 of a consistency derivation the following reduced goals which also must fail.

$$\leftarrow on(Z, X_1, t_2), 4 = X_1 \qquad\qquad \leftarrow on(Z, Y_1, t_2), y_2 = Y_1$$

Let us analyze how the second goal fails within the ACLP computation. Using the first rule for the predicate *on* this will be reduced further to the constraint

$$\leftarrow initially(Z, Y_1, 0), 0 \leq t_2, moved^*(Z, 0, t_2), y_2 = Y_1$$

From the definition of *initially*, this constraint expands to four branches with Y_1 taking one of the values $\{0, 2, 3, 6\}$. All of these branches must fail and one way to do this is to assume (through step `(C3)`) the additional arithmetic constraints $\{y_2 \neq 0, y_2 \neq 2, y_2 \neq 3, y_2 \neq 6\}$.

Note that this is another example where the explicit use of equality allows us to satisfy the integrity constraints by negating this (unification) equalities. At this point, the only values left for the domain of y_2 are the positions $\{1, 5\}$.

The goal obtained when we reduce (step `(C1)`) the above constraint with the second rule of the definition of the predicate *on* is

$$\leftarrow move(Z, Y_1, T), T \leq t_2, moved^*(Z, T, t_2), y_2 = Y_1$$

As the only other assumption in Δ_1 is $move(3, 6, t_1)$ this reduces (via step `(C2)`) further to

$$\leftarrow t_1 \leq t_2, moved^*(Z, t_1, t_2), y_2 = 6.$$

This be made to fail by setting (step `(C3)`) the constraint $t_2 < t_1$ which in fact we already have in the current constraint store C_1.

In a similar fashion, the satisfaction by the original assumption of $move(3, 6, t_1)$ of the second integrity constraint, requiring that a position is clear before we can move a block onto to it, will generate another new assumption of $move(5, y_3, t_3)$ with $t_3 < t_1$. The consistency checking of this new assumption will impose in the same way similar constraints on the domain variables y_3 and t_3. In addition, other constraints linking these with the domain variables y_2 and t_2 of the move of block 4 will be set. For example, from the first integrity constraint, that block 5 must be clear when we move it, we get the following reduced goal that must fail:

$$\leftarrow move(Z, Y_1, T), T \leq t_3, moved^*(Z, T, t_3), 5 = Y_1$$

Using the assumption $move(4, y_2, t_2)$ in Δ this reduces (step (C2)) further to

$$\leftarrow t_2 \leq t_3, moved^*(4, t_2, t_3), 5 = y_2.$$

One way to fail on this is to set (step (C3)) into the constraint store the further domain $t_3 < t_2$ or the constraint $5 \neq y_2$ or the disjunctive constraint $(t_3 < t_2) \vee 5 \neq y_2$. Similar constraints will be set from the interaction between the three moves that are now present in computed set of assumptions. Note that consistency goals that have been reduced by the earlier move assumptions in Δ are remembered by the computation (in Δ^* - see case (C2) of the ACLP proof procedure) and will be reduced further by this new assumption of $move(5, y_3, t_3)$. For example, the following goal reduced from the first integrity constraint by the assumption $move(4, y_2, t_2)$ (and remembered in Δ^*)

$$\leftarrow move(Z, Y_1, T), T \leq t_2, moved^*(Z, T, t_2), 4 = Y_1$$

will be reduced further by the new assumption of $move(5, y_3, t_3)$ to

$$\leftarrow t_3 \leq t_2, moved^*(5, t_2, t_3), 4 = y_3.$$

This will then give the new constraint $(t_2 < t_3) \vee (4 \neq y_3)$ in the constraint store.

In this way, after the full reduction of all the integrity constraints and the goals accumulated in Δ^* several other constraints will be set, in the global constraint store, on the naming variables of the positions and times of the three assumed moves in $\Delta = \{move(3, 6, t_1), move(4, y_2, t_2), move(5, y_3, t_3)\}$. Any solution of the final constraint store giving ground values to the naming variables will constitute an actual plan to reach the final state.

As mentioned above, the final set of assumptions Δ will also contain the NAF assumption $moved^*(3, t_1, t_f)$. Such NAF assumptions express the fact that block 3 should not be moved from its position after the time t_1. If, for example, Δ needed to contain another move for block 3 e.g. $move(3, y, t)$ then the NAF integrity constraint in IC can be satisfied only by requiring that either $t < t_1$ or that $t > t_f$. As this second constraint is not consistent with the constraint store this means that $t < t_1$ will be set i.e. this new move of block 3 will be forced to occur before its move onto block 6 thus ensuring that block 3 will remain on block 6, as required by the final state.

3.2.3. General Features of the ACLP Computation

The above simple example exposes several of the general features of the ACLP computation. Here we will examine further the computation of abductive plans in ACLP and indicate from this the important general computational and search features of the ACLP framework and system.

Let us assume therefore that a new hypothesis $move(b, y, t)$ with both y and t possibly non-ground naming (CLP domain) variables has been generated. Given this new move and any other, $move(b', y', t')$, where similarly y' and t' can also be non-ground, already assumed in the Δ, these two assumptions can "interact" with each other via the integrity constraints in several ways. Consider for example the first integrity constraint written as follows

$$\leftarrow move(b, y, t), move(b', y', t'), t' \leq t, b = y', moved^*(b', t', t)$$

where we have unfolded the definition of the predicate *on* (and rearranged the order of the conditions).

We can see from this form of the integrity constraint that for any such pair of move assumptions there are three ways to make this goal fail (and thus satisfy this constraint) corresponding to the three conditions left in this goal after we resolve away the two move conditions. One way is to set a time constraint $t < t'$, the other to set a space constraint $b \neq y'$ and the third way is to generate a new move for block b' between the times t' and t.

The order in which we choose to apply these options in the ACLP search for a solution can vary. In general, the ACLP search chooses first the options of setting new CLP contraints rather than generating new assumptions. Between setting a time or a space constraint the order is not that crucial as it is possible instead of choosing one of these to set a disjunctive constraint of the form $(t < t') \vee (b \neq y')$ in the constraint store effectively transfering this choice to the constraint solver.

The third option of failing on the $moved^*(b', t', t)$ condition is in fact not the last to be tried but the first in the following sense. The ACLP computation will first try to see if the satisfaction of the integrity constraint can be ensured by checking that it can prove the goal of $moved(b', t', t)$ using only the existing assumptions in Δ and maybe setting extra domain constraints on the time and space variables of these moves. In other words, if we already have a move of the same block b' earlier than the move $move(b', y', t')$ under consideration then there is no need to set any restrictions concerning the later move. If we do not have such a suitable assumption already in the Δ then the possibility of failing by generating a new assumption is delayed as the last choice. This shows one of the ways in which ACLP can *reuse* assumptions already in the current Δ to help achieve more minimal solutions or plans as in this case.

The *minimality* of the abductive solution is also helped indirectly by the least commitment strategy on the abducible assumptions that the ACLP computation adopts. We have seen, in the example above, that although it is possible for the first move of block 5 to go onto block 4 (or for the move of block 4 to go onto block 6) which will then require an additional move to free 4 this does not happen. This is because we can, with the use of setting constraints, avoid committing to a particular destination or time of the moves but leave all possibilities open restricting these (through the constraints) relative to other moves that we have or will be needed later in the computation.

Similarly, consider that we have another place on the table (position 7) and that our final state required also that block number 5 should be moved after the move of block 3 onto 6 (e.g. that 5 should be on top of 3 in the final state). Then again because of the least commitment on the abducible assumptions the move of 4 will not be placed onto block 5 as before but will occupy the new position 7 so that it is then free to move later as required. This is done because the later move of block 5 sets the constraint $y_2 \neq 5$ on the domain variable y_2 of the move of block 4. Here also note that, as described above, if the first move of block 5 that frees block 6 could be identified with the later move of 5 then this will be done thus saving a move from the computed plan.

Even if we did not have the extra position on the table, but we still required the later move of 5, then with the use of the disjunctive constraints, that allow us not to commit on where to set the necessary (relative) restrictions on the space and time attributes of the move assumptions, ACLP will compute a solution where block 4 is moved (to position 1) before block 5 is moved (onto block 4) so that block 5 is free for the later move.

Summarizing, the computation in ACLP has the following three important characteristics

- automatic reduction of the high-level specification
- least-commitment search strategy
- reuse of existing hypotheses and delay of the generation of new hypotheses

on which it relies for its effectiveness and for the quality (minimality) of its computed solutions. We point out again that the interface with constraint solving is crucial for achieving these characteristics.

For particular problem domains we may wish to add some control in the computation either to improve the effectiveness of the computation or the quality of the solution. This can be done easily within the high-level representation framework of ACLP by encoding suitable information in the program P of the abductive theory $\langle P, A, IC\rangle$ representing our problem. Such extensions of the theory can be done in a modular way, as the separation of the representation into P, for the basic model of the problem, and IC, for the major part of the validity requirements of the solution, means that we can refine P to improve the quality of the solutions without affecting the validity which is expressed by IC.

3.2.4. Experiments

In this section, we present some experiments that we have carried out on the blocks world planning problem in order to test the computational performance of the ACLP system and to illustrate its high-level expressivity and flexibility.

Increasing the size of the problem

In the first set of experiments we test the performance of ACLP as we increase the number of blocks in the planning problem. The initial and final configurations are randomly generated starting from 15 blocks with a step of nine. The total number of positions available is fixed to be equal to the number of blocks in the problem plus one third of this number. In other words, the number of positions on the table is one third of the number of the blocks. Table 3.1 shows the execution time (on a SUN ULTRA with 64Mb RAM) and the number of moves required for an indicative set of randomly generated problems with different size. In these experiments the system was asked to find a plan by building the towers of teh final state in a horizontal fashion. It should also be mentioned that in our tests some problems were generated where the system was unable to find a solution. The system was also able to find within times of less than 1500sec solutions to problems of bigger size but where the ratio of the available positions on the table was increased from one third to one half. These times are comparable with the execution times of solving such problems directly in ECLiPSe as reported in [12] especially when we bear in

Performance Measurements				
Blocks	time (in secs)	# of moves	time (in secs) (ground Y)	# of moves (ground Y)
15	3.7	23	9.57	37
24	15.19	34	59	85
33	40.69	49	136.15	116
42	69.5	58	85.96	108
51	325.97	74	308.38	173
60	345.52	87	239.45	162

TABLE 3.1. Performance on Randomly Generated Problems

mind that ACLP is built on top of ECLiPSe and consequently some extra time is needed just to run the code of the ACLP meta-interpreter on top of ECLiPSe.

The last two collumns of the table 3.1 show the performance of ACLP when there is an early commitment on the destination position Y of the assumed moves. These results show how the quality of the solutions (ie. the number of moves required for the plan) is improved in ACLP when we exploit fully its ability to constrain the terms of the abducible assumptions rather than ground them. This least commitment on the assumptions allows their re-use in the computation and hence helps to improve the solution.

Increasing the Complexity of the problem

In this set of experiments we studied the ability of our system to represent new requirements on the planning problem testing the effect on the performance of the planner for these extented planning problems.

We present here an example of this study based on a planning problem that uses 24 blocks. An extra requirement on this core planning problem that "no moves are allowed between a certain time period e.g. (2,8)", would be represented directly in $ACLP$ by the extra integrity constraint:

$$\leftarrow \texttt{move(X,Y,T)}, \texttt{T} > \texttt{2}, \texttt{T} < \texttt{8}. \tag{1}$$

The effect of this on the the planner is shown in Table 3.2 in the second column named $Constr1$ with as expected no significant change on the performance of the underlying core problem shown in the first column of this table.

An alternative new requirement to the problem that was examined states that: no block is allowed to be moved on a specific block (or a set of blocks) between a given time period. For example, if this specific block is block number 1 and the given period is (10,14) then this would be represented by the extra integrity constraint in IC

$$\leftarrow \texttt{move(X,Y,T)}, \texttt{Y} = \texttt{1}, \texttt{T} > \texttt{10}, \texttt{T} < \texttt{14}. \tag{2}$$

The effect of this is shown in the column named $Constr2$ of Table 3.2.

Consider now the requirement that for two specific blocks whenever we move the first block before the other then these two moves must be separated by (at least) a specific time period. Again as an example if these blocks are blocks number 20 and

Performance Measurements (in secs/ # of moves)						
Problem	Standard Conf.	*Constr*1	*Constr*2	*Constr*3	*Constr*4	*All*
24 Blocks	15.19/34	15.57/34	16.29/37	15.60/34	35.28/38	35.33/38

TABLE 3.2. Performance as the Complexity of the Planning Problem Increases

number 2 and the separation time period is 12 units then the following constraint would be added in IC

$$\leftarrow \texttt{move}(20, Y_1, T_1), \texttt{move}(2, Y_2, T_2), T_1 < T_2, T_2 - T_1 < 12. \qquad (3)$$

Assume now that we are given a new requirement that whenever we move a certain block then all its related blocks, defined in some problem specific way, must also be moved within a specific time period. An example of this integrity constraint is given below and its effect on the performance of the system is shown in collumn $Constr4$ of Table 3.2.

$$\texttt{moved}(B, T_1, T_1 + 6) \leftarrow \texttt{move}(16, Y_1, T_1), \texttt{related}(B, 16). \qquad (4)$$

The effect of all of the above constraints is shown in the last column of Table 3.2. In general, these experiments show that $ACLP$ is robust in accommodating such extra requirements. In summary, these experiments show that:

1. The performance of ACLP is comparable to its underlying CLP system despite the fact that it is implemented on top of this.
2. ACLP is robust in accommodating extra or changing requirements on the problem.

3.3. Job-Shop Scheduling in ACLP

Job-shop scheduling is defined as the problem of assigning in time n jobs on m machines where each job is a request for the scheduling of a set of tasks with a particular order. Each job has a specified *release time* after which its execution can start and has to complete its work before a specified *deadline*. In addition, the schedule must satisfy other basic constraints such as the *precedence* constraints that define in which order the different tasks of a job should be carried out and the *capacity* constraints that prevent resources from being allocated to more tasks than they can process at one time (*resource capacity*). Other constraints specific to the particular application may be needed, making the problem more difficult to solve.

In ACLP the precedence and resource capacity constraints can be represented with the following integrity constraints in IC:

$$\leftarrow \texttt{start}(J, T1, R1, S1), T2 \texttt{ is } T1 - 1, \texttt{start}(J, T2, R2, S2),$$
$$\texttt{duration}(T2, D2), S1 < (S2 + D2).$$

$$\texttt{decoupled}(T1, S1, T2, S2) \leftarrow \texttt{start}(J1, T1, R, S1),$$
$$\texttt{start}(J2, T2, R, S2), T1 \neq T2.$$

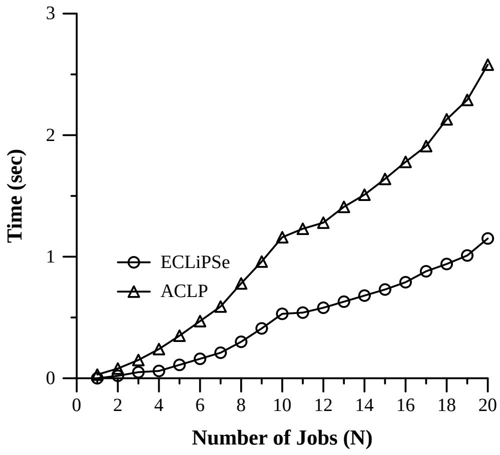

FIGURE 3.2. One solution, job shop scheduling problem using ECLiPSe and ACLP

where $start(J, T, R, S)$ denotes that task T of job J starts execution at time S on resource (machine) R, and is an abducible predicate. The program P of the ACLP job-shop scheduling theory is a simple representation of the basic features of the problem that generates the abductive hypotheses $start(J, T, R, S)$ for each job and task from some top level goal. It also contains the definition of auxiliary predicates that are used in the integrity constraints e.g. $decoupled/4$ which states that two jobs cannot overlap when they run on the same resource.

3.3.1. Experiments

In this section we present a second set of experiments on job-shop scheduling problems.Their purpose is to test further the computational viability of $ACLP$ by comparing the performance of $ACLP$ with the underlying CLP language on which it is implemented and to illustrate again the expressive power of this system.

Increasing the size of the problem

The core scheduling problem that was used in our experiments has 20 jobs with 5 tasks each, (hence a total of 100 tasks) sharing 10 resources. It was constructed from the set of problems defined in [50] by putting together two of these problems. The ACLP implementations of this problem were tested against corresponding implementations directly in ECLiPSe.

Figure 3.2 displays the performance of these two implementations with respect to the size of the job-shop problem ranging from 1 job (5 tasks) to 20 jobs (100 tasks). This figure indicates that the performance of the ACLP implementation is comparable with that of ECLiPSe, since in every size of the problem, the ECLiPSe implementation is about two times faster than the one on ACLP. This is a constant factor of difference that does not increase with the size of the problem. It is worth noting here that the ACLP system is built on top of ECLiPSe and consequently a big fraction of the performance loss is due to the penalty paid to run the code of the meta-interpreter.

In the ACLP system, constraints can be represented directly from their natural specification in a declarative statement. Using a standard CLP system, it is often first necessary to extract the constraints from the problem specification, and then to formulate this set of constraints appropriately. In the ACLP framework this (programming) time consuming process has been reduced considerably. The penalty paid concerning the performance of the system is due to the fact that now during the computation, the low level constraints are constructed from the higher level integrity constraints, formulated to a set of constraint goals and then transmitted to the underlying specialized constraint solver of ECLiPSe.

Increasing the Complexity of the Problem

In this set of experiments we considered the addition of new constraints to the job-shop scheduling problem. We studied the ability of the ACLP system to represent these constraints and compared both implementations (the ACLP and the ECLiPSe version) not only according to their performance results but also in their ability to represent these constraints and their flexibility in changing the problem requirements.

Assume that we are given an extra requirement on the initial core job-shop scheduling problem which says that: for a specific task τ_0, if this starts after a specific time t_0 then this task has to be executed last (i.e. no other task can start execution after the task τ_0). This is represented in our system by the single integrity constraint (5) (for the specific case of $\tau_0 = 14$ and $t_0 = 21$)

$$\leftarrow \texttt{start}(1,14,\texttt{R0},\texttt{S0}), \texttt{start}(\texttt{J},\texttt{T},\texttt{R},\texttt{S}), \texttt{T} \neq 14, \quad \texttt{S0} > 21, \texttt{S} > \texttt{S0}. \tag{5}$$

This constraint was implemented also in the ECLiPSe language. The performance of each approach is presented in Table 3.3 on the second column named *Constr*5, whereas the first column presents the results on the underlying core problem of a fixed size . The ACLP version is still slower than the ECLiPSe version, but without any significant overhead.

An alternative new requirement to the problem that was examined states that: after the end of a specific task τ_1 and for a specific resource r_1, no other task can start execution on that resource before the end of a time interval t_i. This is represented in the ACLP system by the single integrity constraint (6), (for the specific case of τ_1=23, $r_1 = 1$ and t_i =20). The performance of the two implementations is shown in the column named *Constr*6 of the Table 3.3.

$$\leftarrow \texttt{start}(\texttt{J},\texttt{T},1,\texttt{S}), \texttt{start}(2,23,1,\texttt{Sd}), \texttt{duration}(23,\texttt{Dd}), \quad \texttt{S} > \texttt{Sd}, \texttt{S} < \texttt{Sd} + \texttt{Dd} + 20. \tag{6}$$

We point out that in both these cases of adding the constraints (5) and (6) a considerable programming effort was required in order to achieve these execution times by ECLiPSe.

Consider now the requirement that if at least two jobs start their execution in a specific time interval (t_s, t_e) using a specific resource r_i , then no other task can start execution in the interval $(t_e, t_e + t_d)$. The resource r_i must have a rest period. This is represented in ACLP by the integrity constraint (7), where $r_i = 0$, (t_s, t_e) $= (0, 32)$ and t_d =10. The execution time of the ACLP system is shown in the column named *Constr*7 of Table 3.3. Due to the much needed effort to implement

Performance Measurements (in secs)						
System	Standard Conf.	$Constr5$	$Constr6$	$Constr7$	$Constr8$	$Constr9$
$ACLP$	0.90	1.04	0.94	1.34	3.45	1.62
ECL^iPS^e	0.42	0.68	0.42	*	*	*

TABLE 3.3. Performance Measurements as the Complexity of the Problem Increases

this directly in ECLiPSe we did not carry this out!

$$\begin{aligned}\leftarrow \texttt{start(J,T,0,S)}, \texttt{start(J1,T1,0,S1)}, \texttt{start(J2,T2,0,S2)},\\ \texttt{T1} \neq \texttt{T2}, \texttt{0} < \texttt{S1}, \texttt{S1} < \texttt{0} + \texttt{32},\\ \texttt{0} < \texttt{S2}, \texttt{S2} < \texttt{0} + \texttt{32}, \texttt{S} > \texttt{32}, \texttt{S} < \texttt{32} + \texttt{10}.\end{aligned} \quad (7)$$

Assume now that we are given a new requirement on the initial problem which says that: if any task τ_i is using the resource r_i then the related with r_i resource r_j has to be idle till the end of τ_i. This is represented in ACLP by the integrity constraint (8) and the performance of its implementation is presented in Table 3.3. Again this constraint was not implemented directly in ECLiPSe.

$$\begin{aligned}\leftarrow \texttt{start(J1,T1,R1,S1)}, \texttt{start(J2,T2,R2,S2)}, \texttt{T1} \neq \texttt{T2},\\ \texttt{related(R2,R1)}, \texttt{duration(T2,D)}, \texttt{S2} < \texttt{S1}, \texttt{S1} \leq \texttt{S2} + \texttt{D}.\end{aligned} \quad (8)$$

The previous constraint required that no task could start on resource $R1$ at a time when the related resource $R2$ is busy. Consider now the "dual" constraint requiring that no task can begin on resource $R1$ at a time when the related resource $R2$ is idle. In other words, if a task starts at time T on resource $R1$ then the related resource $R2$ must be working at this time T. This requirement is represented in ACLP by the following integrity constraint (9):

$$\texttt{working(R2,S)} \leftarrow \texttt{start(J,T,R1,S)}, \texttt{related(R2,R1)}. \quad (9)$$

where `working/2` is defined in the program P of the ACLP theory as follows:

```
working(R,S) :-
        select_task(Ja,Ta,R),
        start(Ja,Ta,R,Sa),
        duration(Ta,Da),
        S > Sa, S < Sa + Da.
```

The effect of this constraint is that whenever a task T is scheduled on resource $R1$ the system dynamically schedules in the consistency phase of task T another task Ta on the related resource $R2$ to ensure that this constraint is satisfied. The execution time of this constraint is shown in the last column of Table 3.3. Note that as mentioned above the asterix in this table shows that we do not have a time for the ECLiPSe case as these constraints required a considerably larger effort to be implemented directly in ECLiPSe.

Changing the Requirements

Another important feature of the ACLP framework is that it facilitates the adoption of new requirements (eg. rescheduling) with minimal changes to the existing solution. There are three reasons that help ACLP in this process of recomputation of a new solution. Firstly, as we saw in the above experiments, the expressiveness which is provided in the language allows the easy and direct representation of any change in the domain of application. The other reason, which is a consequence of the high-level modeling environment of ACLP, is the clear separation of the validity of a solution by the representation of ACLP in the integrity constraints of the ACLP theory $\langle P, A, IC \rangle$ for the problem. Thus one can change the validity requirements on the solution locally in IC, without any changes required to the overall model of the problem in P. The third reason comes from the inherent ability of abduction to reason with a non-empty initial set of assumptions (partial solution) which enables us to use the old solution and recalculate only what is affected (rendered inconsistent) by the changes in the requirements. This helps to achieve minimal changes in the existing solution.

In this section we present and test this feature of ACLP by studying the problem of rescheduling. To illustrate this consider an example where after we have obtained a solution to a given job-shop problem a new requirement is added which says that: resource R_5 is temporarly out of order (e.g. between the time points 12 and 20). Our new task is to satisfy this requirement with minimal changes to the existing solution. This new requirement is easily incorporated in the ACLP representation of the problem by adding the new integrity constraint:

$$\leftarrow start(J, T, 5, S), S > 12, S < 20.$$

One way then to satisfy this new requirement would be to run the program again, as for the computation of the initial solution, i.e. with the same query `aclp-solve( schedule,[],Solution)` for the initial goal of `schedule` and with an empty set of initial hypotheses, to get a new solution. However, this new solution will (probably) vary significantly from the previous solution, and this contradicts our goal for minimal changes. Instead we can run the ACLP query `aclp-solve(schedule, OldSolution,Solution)` giving the (or part of the) old solution as a set of initial hypotheses to get a new solution that would be close to this previous solution.

In general, given a set of changes in the application environment, the task of rescheduling under ACLP can be summarised in the following basic steps :

1. Remove from the old schedule all hypotheses which are known to be affected by these changes[12]. This gives a new partial solution ($OldSolution'$). In fact this first step is optional, helping only in the efficiency, since hypotheses which make the existing schedule inconsistent will be re-tried and removed automatically by ACLP. Note also that several heuristics or strategies can be adopted in order to decide which part (assumptions) of the solution is best to delete first.

2. Add the new requirements (changes). These may be in the form of constraints (as in the example above) or simply as new information in the domain of the application.

[12] This can be done by a separate module (specific to the particular application) that calculates the hypotheses which are affected.

Performance & Quality Measurements			
Problem # tasks - Resource Availability	Initial Solution *secs*	Rescheduling *secs/changes*	Re-Execution *secs/changes*
25 tasks $R4$ unavailable in period (0,20)	0.23	0.27/4	0.23/21
50 tasks $R2$ unavailable in period (20,30)	0.71	1.94/14	0.74/40
75 tasks $R4$ unavailable in period (0,16)	1.03	1.25/5	1.81/25
100 tasks $R7$ unavailable in period (20,35)	1.61	1.95/1	1.67/47

TABLE 3.4. Rescheduling Experiments for Resource Unavailability

3. Re-execute the problem with the set of the hypotheses in step 1 as a given initial set (ie. call: `aclp-solve(schedule, OldSolution′, NewSolution)`).

When a hypothesis (eg. *start(J,T,R,S)*) is needed during the re-execution in step 3, the ACLP system tries first to match this hypothesis with an existing one in $OldSolution'$ and then proceeds into the consistency phase for that hypothesis. Otherwise, if such a match is not possible or is incosistent then the computation proceeds with an abductive phase where a new hypothesis will be generated in the new solution. This means that ACLP gives priority to existing hypotheses and tries to include these in the final solution of the problem thus reducing the changes on the old solution.

A number of rescheduling experiments have been performed with ACLP on the job-shop problems, presented earlier in this section. Table 3.4 shows an example of the results obtained from these experiments.

The first column of the table shows the cpu time of the initial scheduling of a problem with 25, 50, 75 and 100 tasks. Then, for each one of these problems a new requirement, on some resource unavailability, was added and the goal was to satisfy the new requirements with minimal changes on the initial solution. The rescheduling results are displayed in the second column of the table, which gives the cpu time (in seconds) together with the number of changes needed on the existing solution in order to satisfy the new requirements. The third column displays similar information when the program was re-executed without any initial solution in order to solve from the beginning the same problem together with the extra requirement of resource unavailability. The table shows clearly the appropriateness of the above proposed general strategy for rescheduling within the ACLP framework.

Note that the rescheduling times, in the second column of the table, are comparable to those obtained when re-executing the problem, in the third collumn. This occurs because in the general purpose rescheduling module that we are testing the use of any old assumption, from the existing solution, can be done after it is checked for consistency. In other words, even those assumptions which are not affected by the changes are re-checked for consistency. In specific applications, and for specific changes, special strategies can be adopted in which the consistency of a subset of the old assumptions does not need to be re-checked.

Summarizing the results of the above three types of experiments on the problem of job-shop scheduling we can draw, as in the case of planning, the following conclusions:

1. The performance of the ACLP system is comparable with that of the underlying CLP language.
2. The expressive power of ACLP is higher than that of a standard CLP system, with greater flexibility for problems with changing specifications.
3. ACLP facilitates the recomputation with minimal changes of an old solution that has to adapt to changes in the application environment.

4. GENERAL CHARACTERISTICS OF ACLP APPLICATIONS

In this section we report on the main general characteristics of ACLP applications, generalizing from our experience with the planning and scheduling applications described in the previous section and the real-life applications of timetabling [31] and crew-scheduling [43] developed within the ACLP framework. These characteristics can give us general methodological features and guidelines for the better development of abductive ACLP applications to complex problems.

The possibility for a high-level representation of the application domain, with the ability to model directly properties of the problem, is the central characteristic of ACLP applications. The process of abduction, in ACLP, generates solutions in terms of the problem specific abducibles. For each problem we define its own answer holders, the abducible predicates. These have a higher semantic meaning than simply that of a (problem independent) logical variable which is the usual answer holder of Logic Programming and Constraint Logic Programming. The fact that now the solution carriers of our problem are directly represented by the problem specific abducible predicates allows us to express the natural structure of our problem and the properties of the required solution in a more direct and faithful way, from their natural specification. This in turn means that we can now model easily complex domains and more importantly we can accommodate the particular specialized requirements that a specific application might have. Hence when deciding whether an application is suitable or will benefit from the use of abduction and ACLP one needs to look for the following characteristics: complex functionality with specialized needs which are not necessarily fixed but can dynamically change.

The high-level representation offered by ACLP has two further advantages in the development of an application: (i) the development can be modular and (ii) the resulting solution is flexible under changes of the requirements. Let us analyze these a little further.

The main dimension of modularity in abductive applications comes from the fact that we can separate the two issues of validity and optimality of the solution. We can improve the quality of the solution by incrementally refining the model of the problem, represented in the program P of the ACLP theory, without affecting the validity of the solution which is ensured by the integrity constraints IC of the theory. We can also experiment with different design alternatives adopting different strategies to improve the optimality of the solutions. On the other hand, we can change the requirements on the solution, expressed by the integrity constraints IC, without affecting the way that the problem is modeled in the program P.

The basic model of the problem in P can be incrementally refined along two different but inter-related directions. The first possibility is to exploit incrementally natural characteristics of the problem in order to build in P a more informed model. This can result not only in improvements on the quality of the solution but also on the computational effectiveness of the solution. The second possibility refers to the high- level handle that we have on the solution through the abducible predicates and the choice of abducible that we build in the model P of our problem. This choice can be easily implemented to follow either some heuristics, priorities, or algorithm for optimality, thus directly controlling the quality of the solutions. With this we can also control the way a solution is found or build up, where for example we may want the choice of abducible to follow a least commitment principle, or that this choice is only implicit through some reduced constraints on the hypotheses of the solution.

The other major characteristic of ACLP applications is the flexibility they offer under new or dynamically changing requirements. Once we have one complete representation of the problem we can easily experiment with different requirements on the solution, by changing the integrity constraints which specialize the general model to the needs and preferences of a particular case. Furthermore, the built-in characteristic of abduction to be able to reason with existing hypotheses/solutions facilitates the task of recomputing a solution, or more precisely the task of adapting an old existing solution to accommodate new requirements that appear in time (e.g. for the case of rescheduling). Such adaptations can be done in a natural way with minimal changes to the existing solution, preserving parts of the solution that should not be affected. Also natural requirements and strategies for the recomputation of the solution can easily be accommodated within the high-level modeling environment of ACLP.

5. RELATED WORK

The work of this paper relates to other work in three different (but related) areas of (i) constructive abduction, (ii) constraints in abduction and (iii) extensions of CLP for more declarative representations.

ACLP provides a general framework for the extension of abduction in LP to constructive abduction. It was developed based on and extending early work in [29] for constructive negation through abduction which in turn was based on the work of [14] for NAF through abduction. Recent work on constructive abduction in Logic Programming can be found in [7, 15, 55].

The abductive proof procedure SLDNFA of Denecker and De Schreye [6, 7] incorporates abduction and negation as failure for the treatment of non-ground abductive goals in normal abductive programs. SLDNFA is an extension of the SLDNF procedure for the cases where an abducible atom is selected in a positive or negative goal. The case where an abducible atom is selected in a positive goal is solved by skolemizing the atom before adding it to the set of assumptions. The introduction of skolem constants raises the need to extend unification and resolution with rules for equality. The case where an abducible atom is selected in a negative goal is what differentiates this procedure from previous related work. Instead of storing all negative literals, for which a failure tree is to be computed, and rebuilding their failure trees each time a new fact is abduced, SLDNFA interleaves the computation of this failure tree with the construction of the assumption set. Then, unification and resolution for negative goals is extended to deal with skolem constants in negative goals. The treatment of integrity constraints in SLDNFA is done by adding, for any integrity constraint ic, the rule $false \leftarrow not\ ic$ to the program and the literal $not\ false$ to the query. Soundness and completeness results for SLDNFA are obtained wrt the three-valued completion semantics for abductive logic programs.

In his work, Teusink [54, 55], presents a transformation from extended logic programs (with strong negation together with NAF) to normal programs. He then proposes a proof procedure for abductive logic programs, which is a generalization of the SLDNFA proof procedure for normal logic programs based on constructive negation instead of NAF. Furthermore, the treatment of non-ground queries is done through the use of equality in the language, instead of using skolemization. This procedure is sound and complete wrt a three-valued completion semantics for abductive logic programs.

The Iff proof procedure of Fung and Kowalski [15] can be regarded as a hybrid of the proof procedure of Console et al.[3] and the SLDNFA procedure of Denecker and De Schreye [6, 7]. In comparison with SLDNFA the Iff procedure has several differences some of which are the following. The predicates in this procedure are defined in if-and-only-if form, it uses explicit rewrite rules for equality, instead of extending unification, and avoids skolemization by using existential quantifiers and free variables. The Iff proof procedure is a rewriting procedure, consisting of a number of inference rules (eg. unfolding, propagation, rewrite rules for equality, logical simplifications etc), each of which replaces a formula by one which is equivalent to it. Given an initial query Q, the procedure constructs an initial formula which is Q conjoined with the set of integrity constraints. Answers can be obtained, by exhaustively applying the inference rules on Q, from a nonfailure leaf note by constructing a substitution for the variables in the query, such that both the resulting instance of the query and the integrity constraints are implied by the extended set of definitions. The procedure is proved to be sound and complete relative to the three-valued completion semantics.

Comparing these approaches with ACLP there are several similarities as well as differences. The naming technique used in this paper for teh ACLP proof procedure is very similar to techniques of skolemization done by Shanahan in [51], Missiaen et al in [44], and in the original SLDNFA procedure [6]. Also it is closely related to the distinction between universal and existential variables done in the IFF procedure [15]and the later work on SLDNFA [7]. One of the major differences is the fact that these approaches deal explicitly with the equality and disequality constraints of extended unification, whereas ACLP relies on an integrated external constraint solver. In this way ACLP easily allows other constraints e.g. inequality and logical constraints. Another difference is the treatment of integrity constraints where ACLP is more focussed and goal-oriented in handling these, though for general forms of integrity constraints which cannot be transformed (classically) into denial form ACLP will need to apply a similar technique as SLDNFA. One other difference is the treatment of NAF which in ACLP is done through abduction, extending the work of [14], in the same way as any other type of abducible whereas in these other approaches a separate treatment of NAF is employed.

Despite this, if we examine closely the computational behavior of these approaches one finds strong similarities between these and ACLP in their treatment of NAF. For example, SLDNFA and IFF have computation steps which are analogous to the ACLP steps for dealing with NAF. More specifically, in IFF a negative literal $\neg A$ is replaced by the rewrite goal "false $\leftarrow A$", which is analogous to the consistency derivation step of ACLP, for A^*, under the canonical integrity constraint $\neg(A \wedge A^*)$. Similarly, in SLDNFA the separation of goals to positive and negative is used so that a positive goal $\neg A$ is transformed into a negative goal A which must fail for the computation to succeed. Again this is very similar to the treatment of NAF in ACLP through the consistency phase for A^*. In both SLDNFA and IFF it is possible to satisfy a NAF goal by rewriting this to (or deriving) explicit equality goals. These goals are analogous to the equality constraints that ACLP would generate, although ACLP can also generate constraints on other CLP predicates.

All of these approaches use some variant of the completion semantics for NAF whereas ACLP uses the partial stable model semantics [9, 28, 48] or the stable model semantics for a subclass of logic programs on which these two semantics coincide. Note that using the weaker three valued completion semantics these approaches are able to prove completeness results for their computational models. As we have already mentioned at the end of section 2.3, with the stronger semantics that we have adopted for ACLP we are not able to prove completeness. This however may be possible if we adopt the weaker completion semantics as in the above approaches. For a restricted but important class of logic programs which are stratified, acyclic or locally stratified, the partial stable model semantics coincide with other semantics, eg. the completion semantics, and hence the ACLP proof procedure is correct with these semantics as well. In this sense, if we separate out our treatment of CLP constraints, and bearing in mind the similarities between ACLP and these other methods, the ACLP proof procedure comes closer to those of SLDNFA and IFF developed under the completion semantics.

As mentioned above, these works concentrated only on the treatment of equality (and disequality) constraints in order to perform constructive abduction. Several other works have been made recently to integrate abduction with constraint solving in a more general sense. Some of this work is presented below.

Related work in [37, 59] presents a theorem- proving approach which aims to unify Abductive Logic Programming (ALP), Constraint Logic Programming (CLP) and Semantic Query Optimization (SQO). In this new framework, called CALOG, the constraint solver is programmed by means of built-in constraint definitions, augmented with explicit integrity constraints. This use of integrity constraints is similar to constraint handling rules as in [16]. Answers are obtained in this framework by applying one of the following operations to the initial goal which is conjoined with the integrity constraints: unfolding a reducible atom, logical equivalence transformation, propagation and equality rewriting. The CALOG proof procedure is based on the Iff proof procedure of [15] and may be regarded as a generalization of it. CALOG is also related to the SLDNFA procedure of [7] which performs both abductive and deductive reasoning. This framework has been used to study the problem of job-shop scheduling in [56, 58].

Other related work is the extension of the SLDNFA proof procedure [7] with finite domain constraint solving [2, 8]. SLDNFA is coupled with a finite domain CLP solver, called ROPE, as follows: when SLDNFA selects a constraint (equality or disequality, or domain specification) this constraint is given to the ROPE system. SLDNFA-ROPE has been applied to the problem of scheduling the maintenance of the power units in the power plants of an electricity company [8]. This approach has also been applied in the context of proving infinite failure goals [2]. The integration of constraint and abductive reasoning into a viable intelligent scheduling framework is also studied by the Intelligent Scheduling project at the Institute for Robotics and Intelligence Systems (IRIS, university of Alberta).

In most of these related works the integration of abduction and constraint solving is performed at a different level than the one we are studying here. These works incorporate their own constraint solving techniques into the steps of the overall abductive reasoning, whereas in our case the method of constraint solving is not known to the abductive reasoner: constraint solving is done externally to abductive reasoning but linking strongly its results to it. Moreover, the constraint handling methods of these works are often also seen as a special form of abductive reasoning and thus they can be uniformly integrated in the overall problem specific abductive reasoning. Although this perhaps gives a deeper form of integration it is not clear if such a model of cooperation can be computational effective, like the looser model that we have adopted which relies on specialized techniques of constraint solving hidden from the problem specific abductive reasoning.

From the point of view of CLP, our work is related to work that aims to allow a richer language of constraints in CLP, e.g. the work on Constraint Handling Rules (CHR) or Propia in ECLiPSe, in order to express special specific constraints (or requirements) of the particular problem at hand. Our work confirms that abduction in ACLP can provide such a framework of extension for CLP, with a firm semantic basis that is computationally robust. Initial comparison experiments show that the ACLP system is at least as efficient as that of CHR. We also note that for the problem of planning the performance of our abductive approach is comparable to planners developed directly in CLP e.g. [12].

6. CONCLUSIONS AND FUTURE WORK

The computation of abduction in many cases requires specialized constraint solving. In this work we have studied one possible integration of these two paradigms, within the framework of Logic Programming, by proposing an integrated framework, called ACLP, for Abductive Logic Programming and Constraint Logic Programming.

The integrated framework of ACLP has a firm semantic basis and suitably combines the central notions of abductive explanation in ALP and constraint solving in CLP. A general proof procedure for goal satisfaction has been defined, and proved to be correct. In this proof procedure the general pattern of computation of abductive solutions consists of a suitable cooperative interleaving between hypotheses and constraint generation with consistency checking of the abductive hypotheses and constraint satisfaction. The integration of constraint solving has an enhancing effect on the performance of the abductive computation both in terms of efficiency and in terms of the quality (e.g. minimality) of the solutions that the ACLP computation produces.

Constructive abduction with constraint solving has been recently studied by several other works. In most of these the integration of the two is performed by unifying these concepts and their treatment into one. In this paper we study an approach of integration where the two main tasks of abductive reasoning and constraint solving retain their separate identities but cooperate in a non-trivial way to solve goals. From the point of view of CLP, our work is related to work that aims to allow a richer language of constraints in CLP in order to express special specific constraints (or requirements) of the particular problem at hand. Our work confirms that abduction can provide such a framework of extension for CLP with a firm semantic basis that is computationally robust. The high-level modeling environment of abduction offers an extra layer of modularity and flexibility useful for applications with complex and changing requirements.

We have demonstrated, through the solution of the problems of planning and scheduling, that the interface of abduction with constraint solving in ACLP can form an appropriate framework for application problems in Artificial Intelligence. Within ACLP problems can be represented directly from their declarative specifications, thus facilitating the modeling of complex domains, with the ability to accommodate in the representation natural structures or special features of the specific application, which can often lead to better solutions. Empirical results, based on an implementation of the ACLP framework on top of the CLP language of ECLiPSe, show that ACLP is computationally viable, with performance comparable to the underlying CLP framework on which it is built. In addition, our experiments show the natural ability for ACLP to accommodate easily and in a robust way new or changing requirements of the original problem. ACLP thus combines successfully the advantages of modularity and flexibility of the high-level representation afforded by abduction together with the computational effectiveness of low-level specialized constraint solving.

The usefulness of the ACLP framework in formalizing different problems within the areas of planning and scheduling or from other complex (real-life) applications is an important direction for future work, with further experimentation in order to estimate better the quality of the abductive solutions of ACLP. Further work is also needed in evaluating and comparing the results of ACLP when solving problems of adapting an existing solution to new information or requirements (eg. the problem of re-planning or rescheduling).

Another direction for future work is the study of different strategies of interleaving of the abductive and consistency phases, in ACLP, with constraint solving in order to improve the efficiency of the computation. Similarly, different selection rules in the computation can also affect the efficiency of the proof procedure. Of particular interest, is the timing of the selection of the constraint store in the consistency derivations. An important issue for future study is the question to what extent, within a class of applications, known effective methods for solving these problems based on efficient lower-level constraint solving techniques can be exploited or simulated within our framework via a suitable selection of the control parameters in the framework. In this respect, we can study how existing CLP techniques for special forms of constraints or for optimization can be integrated within the computational model of ACLP.

Acknowledgements

We thank Paolo Mancarella, R.A. Kowalski and Francesca Toni for many valuable discussions. This work has been partially supported by the European Union, KIT project no 962109.

REFERENCES

1. G.Brewka, J.Dix and K.Konolige. Nonmonotonic Reasoning: An Overview. CSLI publications, 1997.
2. M. Bruynooghe, H. Vandecasteele, D. Andre de Waal and M. Denecker. Detecting Unsolvable Queries for Definite Logic Programs. Technical Report, Department of Computer Science, K.U.Leuven, Belgium, 1998.
3. L. Console, D. Theseider Dupre and P. Torasso. On the relationship between abduction and deduction. Journal of Logic and Computation, 2(5), pages 661-690, 1991.
4. L.Console, M.L.Sapino, D.Theseider Dupre. The role of abduction in database view updating. Journal of Intelligent Systems, 1994.
5. L.Console, L.Portinale and D.Dupr. Using Compiled Knowledge to Guide and Focus Abductive Diagnosis. Journal IEEE Transactions on Knowledge and Data Engineering, vol.8-5, pages 690- 706, 1996.
6. M.Denecker and D.De Schreye. SLDNFA: an abductive procedure for normal abductive programs. Proc. International Conference and Symposium on Logic Programming, 686-700, 1992.
7. M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for abductive logic programs. Journal of Logic Programming, 1997.
8. M. Denecker, H. Vandecasteele, D. De Schreye, G. Seghers and T. Bayens. Scheduling by "Abductive Execution" of a Classical Logic Specification. Technical Report, Department of Computer Science, K.U.Leuven, Belgium, 1997.
9. P.M.Dung. Negation as hypothesis: an abductive foundation for logic programming. Proc. 8th International Conference on Logic Programming, MIT Press, Paris, 1991.
10. P.M.Dung. An argumentation theoretic foundation for logic programming. Journal of Logic programmimg, Vol. 22, 151-177, 1995.
11. ECLiPSe User Manual. ECRC, Munich, Germany, 1994.
12. A.El-Kholy, B.Richards. Temporal and Resource Reasoning in Planning: the parcPLAN approach. Proceedings of ECAI-96, 1996.
13. K.Esghsi. Abductive Planning with the Event Calculus. Proceedings of the Fifth International Conference on Logic Programming, 1998.
14. K.Eshghi, R.A.Kowalski. Abduction compared with negation by failure. Proc. 6th International Conference on Logic Programming, ICLP89, Lisbon, MIT Press, 1989.
15. T.F.Fung, R.A.Kowalski. The IFF Proof Procedure for Abdcutive Logic Programming. Journal of Logic Programming, 1997.
16. T.Frühwirth. Constraint Handling Rules. In Podelski, A. (ed.): Constraint Programming: Basic and Trends, LNCS910, Springer Verlag, 1995.
17. M.Gelfond, V.Lifschitz. The Stable Model Semantics for Logic Programming, Proc. 5th International Conference on Logic Programming, 1988.
18. J.Jaffar, J-L.Lassez. Constraint Logic Programming. Technical report, Dept of Computer Sc., Monash University, 1986.
19. J.Jaffar, J-L.Lassez. Constraint Logic Programming. Proc. 14th ACM Principles of Programming Languages Conference,POPL87, Munich, Germany, 1987.

20. J.Jaffar, J-L.Lassez, M.J.Maher. A Logic Programming Language Scheme. Logic Programming: Relations, Functions and Equations, D.DeGroot, G.Lindstrom, Prentice Hall, 1986.

21. J.Jaffar, M.J.Maher. Constraint Logic Programming: A Survey. Journal of Logic Programming, Special 10th aniversary issue, 19/20, 1994.

22. J.Jaffar, M.J.Maher. Constraint Logic Programming: A Survey. Handbook of Logic in Artificial Intelligence and Logic Programming 5, pages 591-696, D.M. Gabbay, C.J. Hogger and J.A. Robinson eds., Oxford University Press, 1998.

23. A.C.Kakas, R.A.Kowalski, F.Toni. The Role of Abduction in Logic Programming. Handbook of Logic in Artificial Intelligence and Logic Programming 5, pages 235-324, D.M. Gabbay, C.J. Hogger and J.A. Robinson eds., Oxford University Press, 1998.

24. A.C.Kakas, P.Mancarella. Generalized Stable Models: a Semantics for Abduction. Proc. 9th European Conference on AI, ECAI90, Stockolm, 1990.

25. A.C.Kakas, P.Mancarella. Abductive Logic Programming. Proc. NACLP Workshop on Non-Monotonic Reasoning and Logic Programming, Austin, Texas, 1990.

26. A.C.Kakas, P.Mancarella. On the relation of truth maintenance and abduction, Proc. 1st Pacific Rim International Conference on Artificial Intelligence, PRICAI90, Nagoya, Japan, 1990.

27. A.C.Kakas, P.Mancarella. Database updates through abduction. Proc. 16th International Conference on Very Large Databases, VLDB'90, Brisbane, Australia, 1990.

28. A.C.Kakas, P.Mancarella. Preferred extensions are partial stable models. Journal of Logic Programming, 14(3,4):341-348, 1993.

29. A.C.Kakas, P.Mancarella. Constructive Abduction in Logic Programming. Technical Report, Dipartimento di Informatica, Universit di Pisa, 1993.

30. A.C.Kakas, A.Michael. Integrating abductive and constraint logic programming. Procedings of the Twelfth International Conference on Logic Programming, Tokyo 1995.

31. A.C.Kakas, A.Michael. Timetabling in an Integrated Abductive and Constraint Logic Programming Framework. Proc. of the First Int. Conference on the Practice and Theory of Automated Timetabling, ICPTAT-95, 167-177, Edingburgh, (1995).

32. A.C.Kakas, C.Mourlas. ACLP: Flexible Solutions to Complex Problems. Proceedings of Logic Programming and Non-monotonic Reasoning, 1997.

33. K.Konolige. Closure + Minimization Implies Abduction. Proceedings of PRICAI90, 1990.

34. R.A.Kowalski. Problem and promises of computational logic. Proc. Symposium on Computational Logic, Lloyd ed., Springer Verlag Lecture Notes in Computer Science, 1990.

35. R.A.Kowalski, F.Sadri. Knowledge representation without integrity constraints. Technical report, Department of Computing, Imperial College, London, 1988.

36. R.A.Kowalski, M.Sergot. A Logic-Based Calculus of Events. New Generation Computing, vol.4, 1986.

37. R.A. Kowalski, G. Wetzel and F. Toni. A Unifying Framework for ALP, CLP and SQO. Technical report, Department of Computing, Imperial College, London, 1996.

38. J.W. Lloyd Foundations of Logic Programming. Springer Verlag, 1987.

39. J.W. Lloyd, R.W. Topor. A basis for deductive database system. Journal of Logic Programming, Vol 2, pp. 93-109, 1985.

40. M.Maher. A transformation system for deductive database modules with perfect model semantics. Theoretical Computer Science, Vol.110, 377-403, 1993.

41. E.Maim. Abduction and constraint logic programming. Proceedings of ECAI92, Vienna, Austria, 1992.

42. T.Menzies. Applications of Abduction: Knowledge-Level Modeling. International Journal of Human Computer Studies, August, 1996.

43. A., Michael. Abductive Logic Programming and its Application to Scheduling. PhD thesis, University of London, submitted, 1998.

44. L. Missiaen, M. Bruynooghe and M. Denecker. CHICA, A planning System Based on the Event Calculus, Journal of Logic and Computation, Vol 5, no. 5, 579-602, 1995.

45. R.Reiter. On integrity constraints. Proc. 2nd Conference on Theoretical Aspects of Reasoning about Knowledge, Moshe Y.Vardi ed., Pacific Grove, California, 1988.

46. D.Poole, R.G.Goebel, Aleliunas, Theorist: a logical reasoning system for default and diagnosis. The Knowledge Fronteer: Essays in the Representation of Knowledge, (Cercone and McCalla eds), Springer Verlag Lecture Notes in Computer Science, 1987.

47. T.Przymusinski. On the declarative and procedural semantics of logic programs. Journal of Automated Reasoning, 5, 167-205, 1989.

48. D.Saccà, C.Zaniolo. Stable models and non determinism for logic programs with negation. Proc. ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1990.

49. T. Sato, Completed logic programs and their consistency. Journal of Logic Programming, Vol. 9, pp. 33-44, 1990.

50. N.Sadeh. Look-Ahead Techniques for Micro-Opportunistic Job Shop Scheduling. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1991.

51. M.P.Shanahan. Prediction is Deduction but Explanation is Abduction. Proceedings of IJCAI89, 1989.

52. M.P.Shanahan. Robotics and the Common Sense Informatic Situation. 12th European Conference on Artificial Intelligence, Hungary, 1996.

53. M.P.Shanahan. Event Calculus Planning Revisited. Proceedings of the 1997 European Conference on Planning (ECP 97), 1997.

54. F.Teusink. Using SLDFA-resolution with abductive logic programs. ILPS93 post-conference workshop on Logic Programming with Incomplete Information, 1993.

55. F.Teusink. Logic Programming and Non-Monotonic Reasoning. Ph.D Thesis. Department of Computing, University of Amsterdam, 1996.

56. F.Toni. A theorem-proving approach to job-shop scheduling. Technical Report, Imperial College, 1994.

57. P.Van Hentenryck. Constraint Satisfaction in Logic Programming. Logic Programming Series, MIT Press, 1989.

58. G. Wetzel. Scheduling in a New Constraint Logic Programming Framework. MSc. thesis, Imperial College, London, 1994.

59. G., Wetzel. Abductive and Constraint Logic Programming. Ph.D. thesis, Imperial College, London, 1997.

A. PROOF OF THEOREM 2.3

Theorem: *Let $\langle P, A, IC\rangle$ be an ACLP theory where P is a definite program and the integrity constraints in IC are of restricted form (as in definition 2.11). Let also the sequence $(\leftarrow G, \{\}, \{\}), \ldots, (\leftarrow C[]\{\}\ ., \Delta, \Delta^*)$ be a finite (successful) abductive derivation. Define $\Delta_{rn} = Rn(\Delta \cup \exists XC)$, where X is the set of free variables (not naming variables) appearing in the resulting constraint store C. Then Δ_{rn} is an abductive explanation for the goal G.*

PROOF. We consider grounding valuations g for Δ_{rn} that satisfy the constraints of the computed constraint store C and ground the naming variables appearing in the abducibles giving a set of ground abducibles Δ_g. We denote by C_g and G_g the constraint store and goal obtained by applying this grounding on C and G respectively.

The definitions ?? of abductive explanations require us to show that there is at least one such grounding (i.e. grounding g for which C_g is solvable) and for each such grounding the following hold :

1. $M^{\Re}_{P\cup\Delta_g} \models G_g$ and
2. $M^{\Re}_{P\cup\Delta_g} \models IC$

The existence of one consistent grounding of C follows directly from the fact that the computation starts with a consistent (or empty) constraint store and this is updated only through `(A1)`, `(A2)`, `(A3)` or `(C3)` of the ACLP proof procedure. In each of these steps it is required that the resulting set of constraints is solvable, otherwise the proof procedure fails. Hence the final constraint store C is solvable and therefore there exists at least one valuation (grounding) of C, eg. C_g, which is consistent.

Consider now a grounding valuation g with C_g consistent and Δ_g, G_g such that $\Delta_g \models_{\Re} \Delta_{rn}$. We need to show (1) and (2) above.

(A) To show (1) we will show that a successful CLP derivation in $P \cup \Delta_g$ can be constructed for the goal G_g. We follow the selection rule of the ACLP abductive derivation for G. Suppose that a some stage the literal L_j is selected. There are three cases.

`(1a)` L_j is not an abducible. Then we apply the same resolution step as the one applied in the ACLP derivation (step `(A1)`) by the ACLP procedure. Note that this step is an ordinary CLP derivation step.

`(2a)` L_j is an abducible predicate, which in the ACLP proof procedure resolved (step `(A2)`) with a member, $a(n)$, of the set of assumptions Δ. Let L_j have the form $a(X)$, where X is the (possibly empty i.e. L_j is a ground abducible) set of variables appearing in the abducible a. In the CLP derivation that we are constructing we resolve L_j with the ground abducible, $a(g(n))$, where $g(n)$ denotes the ground constants to which the variables n are mapped under the grounding valuation g, that we get from $a(n)$ when we apply the grounding g to Δ_{rn}. Note that the constraint(s) $\{X = g(n)\}$ will be added to the constraint store C' of the constructed CLP derivation.

(3a) L_j is an abducible predicate $a(X)$, which in the ACLP proof procedure was added through step (A3) to Δ (and hence belongs to Δ_{rn}). This must then have appeared in Δ as a new assumption $a(n)$, where n are naming variables for the variables X and the constraints $\{X = n\}$ would have also been added in the constraint store C of the ACLP computation. As in the previous case (2a) in the CLP derivation that we are constructing we resolve L_j with the ground abducible, $a(g(n)) \in \Delta_g$, that we get from $a(n)$ when we apply the grounding g to Δ_{rn}. Again the constraint store C' of the constructed CLP derivation will be augmented with the constraints $\{X = g(n)\}$.

By construction of this CLP derivation at each one of its steps the constraint(s), c', that are added to its constraint store C' belong to the grounding C_g of the final constraint store C. In step (1a), C' is updated by the same constraints (on which the grounding valuation g has been applied) with which the constraint store of the abductive derivation is updated from the corresponding (A1) step of the ACLP procedure. In the other two steps (2a and 3a) from the corresponding ACLP steps we know that $\{X = n\}$ belongs to the final constraint C and therefore $\{X = g(n)\}$ belongs to C_g. Since we know that C_g is consistent the constraint store C' of the constructed derivation is solvable at each step and hence this is a valid CLP derivation. This terminates successfully and hence from the soundness theorem 2.1 of CLP we have that $M^{\Re}_{P \cup \Delta_g} \models G_g$ as required.

(B) We next need to show that for each given integrity constraint, ϕ, in IC, all CLP derivations starting from this constraint (constraints have been restricted to have the form of denials and hence they trivially correspond to a goal) in the program $P \cup \Delta_g$ are finitely failed (FF). Let us assume that this is not so and there exists a successful CLP derivation in $P \cup \Delta_g$ starting from ϕ and terminates with the (solvable) constraint store C_ϕ. From this we will show that the ACLP computation for G must (by rearranging the order of its selection of literals in the goals to follow the selection order of the CLP derivation) contain a corresponding consistency derivation with a branch that arrives at the goal $\leftarrow C''[]\{\}$, such that the constraints C'_g resulting from the local constraint store C' by applying the grounding g to it is equal to C_ϕ.

Without loss of generality we can assume that the constraint ϕ has the form $\leftarrow a(X), R(X)$, where $a(X)$ is abducible and $R(X)$ is the conjunction of the rest of the denial (including any constraints that may be present), and that the first step of the successful CLP derivation in $P \cup \Delta_g$ is a resolution step of $a(X)$ with some ground abducible $a(d)$ in Δ_g. This resolution step will also exist as the first step of the consistency derivation of the abducible a in Δ from which we get $a(d)$ in Δ_g under the grounding g. This starts the corresponding consistency derivation in the ACLP computation. Note that if a in Δ is not ground, i.e. $a = a(n)$ where n are naming variables, then the local constraint store, C'', of this branch of the consistency derivation will be updated with the constraint $\{X = n\}$. If we then apply the grounding g to this local constraint store we will get the constraint $\{X = d\}$, (as $g(n) = d$), which is the same constraint generated by the resolution step in the given CLP derivation.

Suppose that in the CLP derivation the literal L_j is selected, from the resulting goal $\leftarrow \{X = d\}, R(X)$, and resolved in $P \cup \Delta_g$. There are two cases.

(1b) L_j is not an abducible. Then the same resolution step will take place in the corresponding consistency derivation branch of ACLP through step (C1), as in this step L_j is resolved with all possible rules in P for the predicate of L_j. The local constraint store C' of this branch will be updated by the same constraints as those that would be added in C_ϕ of the CLP derivation.

(2b) L_j is an abducible, say $b(Y)$ which is resolved, in the CLP derivation, with a ground abducible $b(d')$ in Δ_g. There are two cases here depending on whether the abducible, $b(n')$, in Δ corresponding to $b(d')$ under the grounding g (i.e. $g(n') = d'$) has been added to Δ before or after the abducible $a(n)$ at the root of these derivations. If $b(n')$ is added before $a(n)$ then in the corresponding consistency derivation branch of ACLP we will have through step (C2) the resolution of $b(Y)$ with $b(n')$ in Δ. The constraint $\{Y = n'\}$ that will be added to the local constraint store C', of this branch of the consistency derivation, will be equal, after applying the given grounding g, to $\{Y = d'\}$ which is the constaint that the CLP derivation adds in its constraint store C_ϕ. Consider now the case where $b(n')$ is added in Δ after $a(n)$ and let $\leftarrow b(Y), R'(Y)$ be the current goal. In step (C2) of the ACLP consistency derivation for $a(n)$, this (goal) denial will be added in the (global) set Δ^* of the computation. Then the corresponding concistency derivation branch in the computation of ACLP is the branch rooted at $b(n')$ (when this is added to the Δ) resolving with $\leftarrow b(Y), R'(Y)$ in Δ^*. Again, as above, the local constraint resulting from this resolution step in ACLP will be updated, after applying the grounding g, with the same constraint added in the resolution step of the CLP derivation to C_ϕ.

Hence following these two steps (1b and 2b) we can find in the ACLP computation a consistency derivation branch that arrives at the goal $\leftarrow C'[]\{\}$ such that the constraints C'_g resulting from this local constraint store C' by applying the grounding g to it, is equal to the final constraint store C_ϕ of the CLP derivation. Therefore, since C_ϕ is solvable so is C'_g i.e. the grounding g is a consistent grounding of C'. On the other hand, the ACLP computation is successful and hence all its consistency derivation branches must end up in failure i.e. all consistency leaf goals must be eliminated. Therefore this corresponding branch of ACLP that arrives at the goal $\leftarrow C'[]\{\}$ must also end in failure. The only way that this can be done is to apply step (C3) of the consistency derivation where this goal is failed (eliminated) by adding to the current global constaint store C_i of the computation a set of constraints $\overline{C}$ such that C_i remains solvable and $C' \cup (C_i \cup \overline{C})$ is unsolvable. But since g is a consistent grounding of the final constraint store C computed by ACLP, g is also a consistent grounding of $C_i \cup \overline{C}$ contained in C and hence g can not also be a consistent grounding of C'. Contradiction.

Therefore, all CLP derivations in the program $P \cup \Delta_g$, starting from a goal $\leftarrow G$ corresponding to any integrity constraint denial ($\phi = \neg G$) in IC, are finitely failed (FF) and so by the soundness theorem of FF CLP derivations (theorem 2.2) we have $M^{\Re}_{P\cup\Delta_g} \models \neg G \equiv M^{\Re}_{P\cup\Delta_g} \models \phi$, as required. □

B. PROOF OF THEOREM 2.4

Theorem: *Let $T = \langle P, A, IC\rangle$ be a general ACLP theory where P is an order-consistent program. Let also the sequence $(\leftarrow G^*, \{\}, \{\}), \ldots, (\leftarrow C[]\{\} ., \Delta, \Delta^*)$ be* *a finite (successful) abductive derivation. Then $\Delta_{rn} = Rn(\Delta \cup \exists XC)$, where X is the set of free variables (not naming variables) appearing in the resulting constraint store C is an abductive explanation for the goal G^* in the positive form $T^* = \langle P^*, A \cup A^*, IC^* \cup I^* \rangle$ of the theory T.*

As in theorem 2.3 we consider grounding valuations g for Δ_{rn} that satisfy the computed constraints C and denote by Δ_g, C_g and G_g^* the corresponding ground abducibles, constraint store and goal respectively. We need to show that

1. $M^{\Re}_{P^* \cup \Delta_g} \models G_g^*$ and
2. $M^{\Re}_{P^* \cup \Delta_g} \models IC^* \cup I^*$.

The existence of at least one such grounding where C_g is solvable is again easily shown for the same reason namely because the proof procedure explicitly checks at each step where this store can be updated that it remains solvable.

(A) To show that (1) holds we can map the extended ACLP computation onto a computation of the abductive proof procedure of Eshghi and Kowalski in [14] for ground abduction of NAF (on which the ACLP proof procedure is based) and appeal to the soundness of this procedure as shown first in [9] (and [10]).

We therefore consider the given successful abductive derivation and trancate from the consistency derivations that it contains the parts that are generated in their initial steps (1) through constraints from IC^* (and Δ_i^*), i.e. we keep only the part of the consistency derivations that are generated via the NAF denial constraints of the from $\leftarrow p^*, p$ in I^*. Then following the same argument steps as in the proof of theorem 2.3 we can map this abductive derivation into a successful abductive derivation for the goal G_g^* of the proof procedure in [14] performing ground abduction for NAF on the program $P_g^* \cup \Delta_g^{'}$. Here P_g^* is the logic program obtained from the rules of the CLP program P^* that are used in the derivation steps `(A1)` and `(C1)` of the procedure and applying the grounding evaluation g on them to eliminate the constraints that they may contain. The set $Delta_g^{'}$ is the subset of $Delta_g$ containing all the non-negative abducibles. Note that the subsidiary consistency derivations of the extended ACLP procedure rooted at a NAF abducible $p^*(n)$ are mapped in the same way into subsidiary consistency derivations of the procedure in [14] for the corresponding ground abducible $p^*(g(n))$. This mapping is straightforward due to the close relationship of the two procedures. Steps `(C1)`, `(C2)` and $(C4_{naf})$ of the extended ACLP proof procedure are mapped directly into `(C1)`, `(C2)` and `(C3)` of the procedure in [14]. The step `(C3)` of the ACLP procedure by construction construction of $P_g^* \cup \Delta_g$ does not apply and these parts of the derivations will be dropped in the mapping.

Given this mapping the soundness of the procedure in [14] ensures that $M^{\Re}_{P^* \cup \Delta_g} \models G_g^*$. In fact, the soundness proof of [9] shows more generally that this result holds when the initial abductive assumptions of a successful abductive derivation is non-empty as in the case of the subsidiary abductive derivations belonging to the given outer abductive derivation. Hence for any subsidiary abductive derivation, of the initial abductive derivation given in the statement of the theorem, for a goal S_s^* we have under the above mapping that $M^{\Re}_{P^* \cup \Delta_g} \models S_g^*$. This will be needed below in part (B) of the proof.

(B) The same result of [9] ensures also that, under the mapping in part (A) of the ACLP computation onto a computation of the procedure in [14], we have $M^{\Re}_{P^* \cup \Delta_g} \models I^*$. In order to show that the integrity constraints IC^* are satisfied we can again follow the same steps as in the proof of theorem 2.3 to show by contradiction that this is the case. We now have an extra case to consider given by the extra step $(C4_{naf})$ of the consistency derivation for the extended ACLP proof procedure. Thus refering to the proof of theorem 2.3 we consider a successful CLP derivation in the program $P^* \cup \Delta_g$ for a goal ϕ where ϕ is a denial constraint in IC^*. The extra case now refers to a selected literal, in this successful CLP derivation, that is of the ground NAF form $p^*(d)$ such that this belongs to Δ_g. This would then mean that the given abductive derivation contains a consistency derivation where $p^*(n)$ (with $g(n) = d$) was selected. From the definition of the step $(C4_{naf})$ this means that there is a successful abductive derivation for $p(d)$ whose computed constraint store and abducible assumptions are subsets of C and Δ computed by the given top level abductive derivation. From part (A) above we have that $M^{\Re}_{P^* \cup \Delta_g} \models p(d)$. This then contradicts the fact that $p^*(d)$ belongs to Δ_g and that $M^{\Re}_{P^* \cup \Delta_g}$ satisfies $\leftarrow p^*(d), p(d)$ in I^*.