ACLP: A case for Non-Monotonic Reasoning

A. C. Kakas, A. Michael and C. Mourlas
Department of Computer Science, University of Cyprus
75 Kallipoleos str., CY-1678 Nicosia, Cyprus
Email: {antonis, antonia, mourlas}@turing.cs.ucy.ac.cy.

Abstract

In this paper we present a non-monotonic reasoning framework, called ACLP, which combines abductive reasoning and constraint solving by integrating the framework of Abductive Logic Programming (ALP) with that of Constraint Logic Programming (CLP). In ACLP the role of non-monotonic reasoning is that of providing an automatic reduction of the high-level problem representation and goals to lower level computational problems of a general problem independent form. It thus provides a bridge between the high-level problem domain properties and domain-independent solving methods. The ACLP framework is applied to the problems of planning and scheduling. We report on some experiments performed in order to test the flexibility and the computational effectiveness of ACLP as compared with the direct use of the (lower level) constraint solving framework of CLP on which this is built. These experiments provide evidence that the non-monotonic framework of ACLP does not compromise significantly the computational efficiency of the solutions thus confirming the computational viability of the framework for the development of flexible solutions to complex applications.

1 INTRODUCTION

Non-monotonicity enhances the expressive power of a representation framework. It allows high-level representations of problems close to their natural specification. The modeling of the problems is thus more direct and faithful. As a result of this many problems in Artificial Intelligence and other areas of computer science have been easily captured within different non-monotonic frameworks. Despite this clear need of non-monotonicity the number of real-life applications of non-monotonic reasoning is very small. One of the main reasons for this is the relative computational inefficiency of solutions based on non-monotonic representations.

It has been argued [Gin96], [Kak95], [Kak96] that one way to address this problem is to view the role of non-monotonic reasoning (NMR) as that of providing an effective automatic reduction of the high-level problem representations and goals to lower level computational problems of a general problem independent form. It is thus proposed that NMR should be appropriately integrated with specialized constraint solvers for lower-level problem independent domain constraints. In this way non-monotonic reasoning provides a bridge between the high-level problem domain properties and domain-independent solving methods.

In this paper we present a non-monotonic reasoning framework which combines abductive reasoning and constraint solving by integrating the framework of Abductive Logic Programming (ALP) with that of Constraint Logic Programming (CLP). This integrated framework, called ACLP [Kak95],[Kak97], combines the high level expressivity of abduction with the effective handling of specialized domain constraints. The non-monotonic framework of ACLP can thus offer solutions of application problems that (i) can effectively combine high-level representation problems with the efficiency of the specialized constraint solving to produce computationally viable solutions and (ii) offer a high degree of flexibility in the development of applications able to respond to their specialized and dynamically changing needs.

ACLP interfaces to abduction constraint solving for lower-level arithmetic constraints in a principal way. This integration has a firm semantic basis and allow us to consider the abductive computation as a process of reduction of the high-level goals and specification of the problem to lower- level problem independent constraints.The integration is non- trivial in the sense that the constraint solver not only solves this final constraint store generated by the reduction but also af-

fects 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.

Two types of experiments have been performed with ACLP on two separate applications, that of planning and the problem of scheduling. 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 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 changes in the problem definition. These experiments provide evidence that the non-monotonic framework of ACLP does not compromise significantly the computational efficiency of the solutions, thus confirming the computational viability of the framework for the development of flexible real-life applications.

The rest of this paper is organized as follows. In the next section we present a short review of the theoretical foundations of ACLP and its computational model. Sections 3 and 4 study how ACLP can be used for A.I. applications focusing, respectively, on the problems of planning and scheduling. Section 5 concludes together with a brief discussion on related work.

2 THE ACLP FRAMEWORK

When developing abductive applications the central notion of an abductive explanation takes the role of the solution to a required goal of our problem. Abduction is then viewed as a process that generates solutions to goals with the abducibles seen as high-level problem specific answer holders. In many applications of abduction these abductive explanations or solutions contain only ground atomic facts on the abducibles. Recently, it has been argued ([Kak95], [Den96], [Fun97], [Kow96]) that it is useful to extend this to allow in the abductive explanations non-ground assumptions of the form $\exists X ab(X)$, where "ab' is an abducible predicate and the variables X are restricted through some set C of constraints defined over an appropriate (often arithmetic) domain. So in effect the hypotheses which are needed in the abductive solutions have the general form $\exists X(ab(X), C(X))$. A grounding of the variables X that satisfies the constraints C gives a specific solution to a corresponding ground goal.

A framework that allows such more general abductive explanations integrating abduction and constraint solving is that of Abductive Constraint Logic Programming ($ACLP$) [Kak95].

Given an underlying framework of $CLP(\mathcal{R})$, an **abductive theory or program** in $ACLP$ is a triple $\langle P, A, IC\rangle$ where:

- P is a constraint logic program in $CLP(\mathcal{R})$ possibly containing negation as failure.
- A is the set of abducible predicates, different from the constraints in $\mathcal{R}$.
- IC is a set of integrity constraints, which are first order formulae over the language of $CLP(\mathcal{R})$ and the program P.

An $ACLP$ theory or program 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.

A **goal**, G, has the same form as the body of a program rule whose variables are as usual understood as existentially quantified. **An answer,** Δ, for a goal, G, consists of a set of assumptions of the form:

- $ab(d)$, where $ab \in A$ and $d \in$ *domain of* $\mathcal{R}$.
- $\exists X(ab_1(X), \ldots, ab_n(X), C(X))$, where $ab_1, \ldots, ab_n \in A$ and $C(X)$ is a set of $CLP(\mathcal{R})$ constraints.

An answer for a goal G is correct if it forms an abductive explanation for G as follows. Given a theory $\langle P, A, IC\rangle$ and a goal G, an answer Δ is a **solution of G** iff there exists at least one consistent grounding of Δ (in $\mathcal{R}$) and for any such grounding (labelling), Δ_g :

- $P \cup \Delta_g \models_{LP} G_g$, and
- $P \cup \Delta_g \models_{LP} IC$

where G_g denotes a corresponding grounding of the goal G and $\models_{LP}$ denotes an underlying semantics of $(C)LP$ possible including negation as failure (NAF).

The details of the corresponding grounding and the formal semantics of the integrity constraints are not important for this paper (see [Kak95]). Also the particular semantics for NAF adopted here will not be crucial for the study of this paper. We just mention that it is natural for us to consider an abductive semantics for NAF as in [Esh89]. Note that the integrity constraints are understood as sentences that must follow from the program together with the abductive hypotheses ($P \cup \Delta_g$) for Δ_g to be a valid set of hypotheses. They express properties that must hold by any set of abducible assumptions or in other words by any solution (or answer) of a goal for this to be accepted.

2.1 COMPUTATION OF ABDUCTION WITH CONSTRAINT SOLVING

Let us now examine how, within this framework of $ACLP$, the intregration of Constraint Solving can affect and enhance the abductive computation of a solu-

tion. An abductive computation in the $ACLP$ framework 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 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.

Example 2.1 *Consider the following ACLP theory and goal* G_0*:*

$P = \{p(X) \leftarrow X > 2, q(X), a(X)$
$q(X) \leftarrow X > 4, X < 10$
together with rules defining the predicate $r\}$,

$IC = \{\neg(X > 8, a(X), r(X))\}$,

$G_0 = p(X)$, *where "a" is the only abducible predicate.*

In an abductive phase, the initial goal G_0 will resolve against the first and second clauses to obtain the new goal $G_1 = X > 2, X > 4, X < 10, a(X)$. We then proceed to abduce $\exists X a(X)$ by adding $a(x)$ (x here is a name for this existential variable treated as a domain variable of the underlying CLP framework) to the assumption set Δ_0. The initially empty constraint store C_0 is extended to the set $\{x > 2, x > 4, x < 10\}$. A consistency phase is then invoked in order to check the consistency of the assumption $a(x)$. This will resolve with the integrity constraint in IC to give a consistency goal $\neg(x > 8, r(x))$. Hence $(x > 8, r(x))$ must fail. One way to achieve this is to make the local set of constraints $\{x > 8\}$ unsatisfiable. This is done by assuming $x \leq 8$ and adding it to the global constraint store C_0. The new constraint store $C_1 = \{x > 2, x > 4, x < 10, x \leq 8\}$ remains satisfiable and reduces to $C_1^{'} = \{x > 4, x \leq 8\}$. Therefore the computation succeeds with the final result $\exists X(a(X), X > 4, X \leq 8)$, which is a solution to the initial goal G_0.

During a consistency phase it is possible for new abductive hypotheses to be generated if these are needed in order to ensure the satisfaction of the integrity constraints. These extra assumptions are generated by subsidiary abductive phases (within the consistency phase) and for each new hypotheses a new subsidiary consistency phase is recursively generated.

In the above example the constraint solver of CLP is used to check for the satisfiability of C_0 and C_1, solve the problem of making the local set of constraints $\{x > 8\}$ unsatisfiable and to reduce C_1 to $C_1^{'}$. In general, the **interface of abduction to the** $CLP(\mathcal{R})$ **specialized constraint solver** is as follows:

- in an abductive phase check the satisfiability of and reduce (or solve) the constraint store
- in a consistency phase expand the constraint store in a way such that (i) it remains satisfiable and (ii) other constraints local to the consistency phase become unsatisfiable.

The constraint solver is essentially a black box, transparent to the abductive theory, that is consulted during the overall abductive computation with the two specific tasks given above. Note that the first task is in fact the standard interface to the constraints 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 constraint store.

Effectively, the overall pattern of the $ACLP$ computation can be seen as a reduction through abductive reasoning of the high-level goal and abductive theory to a set of domain constraints of the underlying CLP. In particular, abduction actively translates the high-level integrity constraints into a set of lower-level domain constraints of CLP which are sufficient to ensure that the abductive hypotheses would satisfy the integrity constraints. From the perpective of the CLP the domain constraints that are needed to solve the high-level goal are generated **dynamically** through an unfolding of the "relevant" part of the program P and integrity constraints IC. Abduction provides the high-level "pattern" of the solution whereas the constraint solver computes the more specific but equally important details of the solution. In this way, we have a framework where abductive reasoning forms, in a principal way, a bridge between the high-level problem specific properties and requirements and problem independent (constraint) solving methods.

An important aspect of the interface of constraint solving to abduction in $ACLP$ is the effect that this has on the control of the search for an abductive solution. In general, when searching for an abductive solution there are two alternative strategies that one can follow. In the first, one generates specific (ground) abductive hypotheses and then tests whether these can be accepted according to the integrity constraints. This in many cases can make the computation very costly even when suitable heuristics and problem specific knowledge is used to guide the generation of the specific hypotheses. The other alternative, adopted here by $ACLP$, is to avoid considering specific assumptions but instead generate constraints on the terms of these hypotheses. The abductive seach, particularly in the consistency phase, can be pruned early by setting extra CLP constraints on the domain variables of the abductive hypotheses. In this way the generation of the abductive

hypotheses follows a **least-commitment** strategy relying on the accumulated set of CLP constraints to implicitly carry together the various specific choices given by the solutions of these constraints.

The ACLP framework has been implemented on top of the ECLiPSe language as a meta-interpreter using explicitly its low-level constraint solver that handles constraints over finite domains (integer and atomic elements). This implementation is based on the sound abductive proof procedure for $ACLP$ proposed in [Kak95] which in turn follows a series of proof procedures [Esh89], [Kak90a], [Kak90b] for abductive logic programming. In the current implementation of ACLP the integrity constraints are restricted to be Horn clauses where the head could be empty and where at least one abducible condition must appear in the body of the constraint.

3 ABDUCTIVE PLANNING IN ACLP

In this and the following section we present the solution of two application problems, namely the problem of planning in this section and the problem of job-shop scheduling in the next section, within the non-monotonic framewrok of ACLP. Several experiments with these solutions are demonstrated in order to test the computational effectiveness of the ACLP system and to illustrate its high level expressivity and flexibility.

Several authors [Esg88], [Kon90], [Sha97] have studied planning as an abductive problem. Indeed, this can easily be formalised through abduction in 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 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 to occur.

A particular planning problem in the domain described by an abductive theory $< P, A, IC >$ can then be solved by adding into the program P a description, S_i, of the specific 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 consist of a set of actions with specific ground instances, Δ_g, satisfying the following conditions:

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

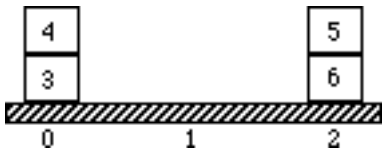

Figure 1: An example initial state

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

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.

As an example let us consider the particular simple but instructive planning domain of the blocks world. This problem can be formalised by the $ACLP$ theory $< P, A, IC >$ with the following general axioms in P:

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

where $on(X,Y,T)$ is a (the only) property predicate meaning that block X is on block or position Y at time T. The predicate $moved(X,T1,T2)$ 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, namely the action of moving a block to a new position. 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 (here read "$\leftarrow$" as "$\neg$"):

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

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

Consider for example the initial configuration as shown in figure 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.

Setting this as our top goal this will be reduced in an (initial) abductive phase of the ACLP computation and will generate the assumption $\Delta_0 = \{move(3,6,t_1)\}$, where t_1 is a new "naming" constant treated as a CLP domain variable. In fact, Δ_0 will also contain a NAF assumption $not\ moved(3,t_1,t_f)$ by treating NAF also through abduction [Esh89]. These assumptions are handled by the abductive computation of $ACLP$ in the same way as the action (move) assumptions. Due to lack of space we will not elaborate on these assumptions further.

A constraint store will also be formed with $C_0 = \{t_1 > 0\}$. Once this assumption is generated a consistency phase starts to ensure that the integrity constraints remain satisfied. This will reduce the two integrity constraints in IC to the following two goals

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

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

$$\leftarrow 0 \leq t_1, not\ moved(4,0,t_1)$$

$$\leftarrow T_1 \leq t_1, move(Z,X_1,T_1), not\ moved(Z,T_1,t_1), \\ 3 = X_1$$

At this stage of the computation the second goal fails trivially as there is no other assumption[1] of a move in the current set of hypotheses Δ_0. The first of these goals can fail, within an ACLP consistency phase, in two ways. One way is to add 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 $not\ moved(4,0,t_1)$. This will result in a new subsidiary abductive phase to try to prove the goal

$$\leftarrow moved(4,0,t1).$$

Using the definition of the predicate *moved* this goal is equivalent to proving

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

The assumption set Δ_0 will then grow to $\Delta_1 = \{move(3,6,t_1), move(4,y_2,t_2)\}$, where y_2, t_2 are new "naming" constants - CLP domain variables. The domain of the variable y_2 is set to be equal to $\{1,2\ldots,6\}$ namely equal to all the available positions. The constraint store will now be $C_1 = \{t_1 > 0, 0 < t_2, t_2 < t_1, y_2 \neq 4\}$. We see here how the consistency checking of one abducible may require 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. The computation will continue in a similar fashion and after simplifying the generated constraint store will produce the final solution:
$\Delta = \{move(3,6,t_1), move(4,5,t_2), move(5,1,t_3), t_2 < t_1, t_3 < t_1, t_2 < t_3\}$, where t_1, t_2, t_3 denote respective times for the three moves.

We emphasize that the computation of plans follows the general computation of $ACLP$ of integrating abductive reasoning with constraint solving with no specialized planning algorithms or control coming from the particular domain of the planning problem. The computation of plans inherits the general characteristics of the computation in $ACLP$ such as

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

and relies on these for its effectiveness. We point out again that the interface with constraint solving is crucial for achieving these characteristics. The soundness of the planner follows directly from the soundness of the abductive computation in $ACLP$.

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 $< P, A, IC >$ 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 validity requirements of the solution means that we can refine P to improve the quality of the solutions without affecting their validity which is ensured by IC.

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. For example, if it is required that some position y' can not be used as a destination for any move during some (open) time period from t_1 to t_2 then we would add the intregrity

[1] In fact, as the effects of our actions are instantaneous we need to safeguard against a move rendering itself inconsistent but this is a detail which is not importnat at this stage of discussion.

constraint

$$\leftarrow move(X, Y, T), T > t_1, T < t_2, Y = y'.$$

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

$$on(X_1, Y_1, T) \leftarrow move(X, Y, T), related(X, X_1),$$
$$fixed - at(X_1, Y_1).$$

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

3.1 EXPERIMENTS

In this section, we present some experiments that we have carried out in order to test the computational viability of the ACLP system and to illustrate its high-level expressivity and flexibility. As mentioned above, the ACLP system [Kak97]is implemented on top of the ECLiPSe [ECL94] language as a metainterpreter using its constraint solver for constraints over finite domains.

These experiments are focused here on the problem of planning (other experiments on job-shop scheduling can be found in the next section) where we test the ACLP system as a planner on the blocks world problem. We will test its performance as well as its flexibility in accommodating changes in the specification of the standard planning problem.

3.1.1 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 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. 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.

Regarding the minimality of the solutions found by $ACLP$ we are not able to estimate well at this stage (we are currently experimenting further on this) how close these solutions are to the optimal solutions. For some problems of small size where we can know what the optimal number of moves is we have encouraging results that our solutions are close to optimal. Another indication of the quality of the solutions can be seen by comparing their mumber of moves with the number of moves required by the solutions found when we commit earlier to the destination position Y of the moves. These solutions as seen in the last two collumns of the table 1 require a significantly larger number of moves.

Table 1: Performance on Randomly Generated Problems

Performance Measurements				
Blocks	time (secs)	# of moves	time (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
69	662.15	101	*	*

3.1.2 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:

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

The effect of this on the the planner is shown in Table 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 blocks are 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 in by the extra integrity constraint in IC

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

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

Table 2: Performance as the Complexity of the Planning Problem Increases

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

these blocks are blocks number 20 and number 2 and the separation time period is 12 units then the following constraint would be added in IC

$$\leftarrow \mathtt{move}(20, \mathtt{Y}_1, \mathtt{T}_1), \mathtt{move}(2, \mathtt{Y}_2, \mathtt{T}_2), \mathtt{T}_1 < \mathtt{T}_2, \quad \mathtt{T}_2 - \mathtt{T}_1 < 12. \tag{3}$$

Assume now that we are given a new requirement that whenever we move a certain block then all its related blocks 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 *Constr*4 of Table 2.

$$\mathtt{moved}(\mathtt{B}, \mathtt{T}_1, \mathtt{T}_1 + 6) \leftarrow \mathtt{move}(16, \mathtt{Y}_1, \mathtt{T}_1), \mathtt{related}(\mathtt{B}, 16). \tag{4}$$

The effect of all of the above constraints is shown in the last column of Table 2. In general, these experiments show that $ACLP$ was indeed robust in accommodating such extra requirements. Our conclusions based on these experiments can be summarized as follows:

1. The performance of $ACLP$ is comparable to CLP (and other approaches) despite the fact that $ACLP$ is implemented on top of CLP.
2. ACLP is robust in accommodating extra or changing requirements on the problem.

4 JOB-SHOP SCHEDULING IN ACLP

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.

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

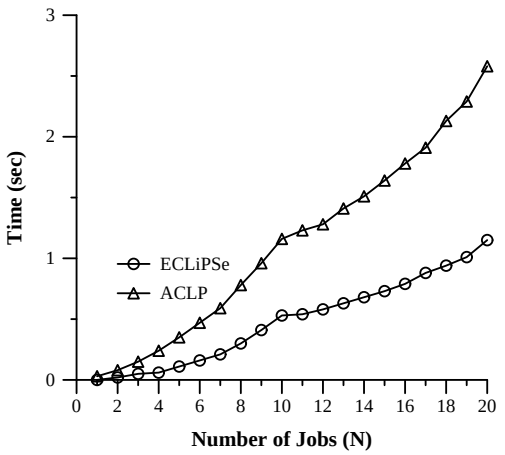

Figure 2: One solution, job shop scheduling problem using ECLiPSe and ACLP

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:

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

$$\mathtt{decoupled}(\mathtt{T1}, \mathtt{S1}, \mathtt{T2}, \mathtt{S2}) \leftarrow \mathtt{start}(\mathtt{J1}, \mathtt{T1}, \mathtt{R}, \mathtt{S1}), \mathtt{start}(\mathtt{J2}, \mathtt{T2}, \mathtt{R}, \mathtt{S2}), \mathtt{T1} = \backslash = \mathtt{T2}.$$

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.

4.1 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 [Sad91] by putting together two of these problems. The ACLP implementations of this problem was tested against corresponding implementation directly in ECLiPSe.

Figure 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.

The main difference between an ACLP based implementation and a standard CLP implementation is concentrated on the way the constraints are expressed. In the ACLP system, constraints can be represented directly from their natural specification in a declarative statement. On the contrary, using a standard CLP system, it is 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 considerable. 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.

4.2 INCREASING THE COMPLEXITY OF THE PROBLEM

In order to illustrate the expressive power of the proposed system for the development of complex applications, 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 if at least two jobs have both started 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 following integrity constraint, where $r_i = 0$, $(t_s, t_e) = (0, 32)$ and $t_d = 10$.

$$\begin{aligned} \leftarrow \mathtt{start(J,T,0,S)}, \mathtt{start(J1,T1,0,S1)}, \\ \mathtt{start(J2,T2,0,S2)}, \mathtt{T1} = \backslash = \mathtt{T2}, \mathtt{0\#} < \mathtt{S1}, \\ \mathtt{S1\#} < \mathtt{0} + \mathtt{32}, \mathtt{0\#} < \mathtt{S2}, \mathtt{S2\#} < \mathtt{0} + \mathtt{32}, \mathtt{S\#} > \mathtt{32}, \\ \mathtt{S\#} < \mathtt{32} + \mathtt{10}. \end{aligned} \quad (5)$$

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:

$$\begin{aligned} :-\mathtt{start(J1,T1,R1,S1)}, \\ \mathtt{start(J2,T2,R2,S2)}, \mathtt{T1} = \backslash = \mathtt{T2}, \\ \mathtt{related(R2,R1)}, \mathtt{duration(T2,D)}, \mathtt{S2\#} < \mathtt{S1}, \\ \mathtt{S1\#} <= \mathtt{S2} + \mathtt{D}. \end{aligned} \quad (6)$$

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:

$$\begin{aligned} \mathtt{working(R2,S)} :- \mathtt{start(J,T,R1,S)}, \\ \mathtt{related(R2,R1)}. \end{aligned} \quad (7)$$

where the predicate `working/2` is defined in the program 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.

In general, such constraints were implemented and added to the basic job shop scheduling formulations within ACLP and CLP. The performance of the ACLP impementation was again slower, by a factor of two, than the pure CLP (ECLiPSe) version. We must point out however that the addition of these constraints in ECLiPSe required a considerable programming effort to the extent that some of these requirements were too

complicated to program directly in ECLiPSe in a relatively short time. In $ACLP$ the only change required is simply to add the above statements to the theory.

Again, as in the case of planning, we can draw from these experiments the following conclusions:

1. The performance of the ACLP system is comparable with that of the underlying CLP language. Much of the execution time of a program written in ACLP is spent for the run of the meta-interpreter code. As a result, a full low-level implementation of the proposed system seems a promising and computational effective programming environment.
2. The expressive power of ACLP is higher than that of a standard CLP system, with greater flexibility in the solution of problems with changing specifications.

5 CONCLUSIONS AND RELATED WORK

We have presented a framework for non-monotonic reasoning performed using abduction. This framework, called ACLP, is based on the frameworks of Abductive and Constraint Logic Programming and integrates abduction with constraint solving. Within this framework the computation of solutions for a variety of different problems follows the same general model of abductive reasoning interfaced non-trivially with constraint solving.

The relation of abduction to non-monotonic reasoning is by now well understood at the theoretical level. In fact, most non-monotonic formalisms (and problems) can be captured by abduction. Given this tight relationship a computationaly viable framework for abduction can also form the basis for an effective computation of non-monotonic reasoning. More importantly, a similar integration of specialized constraint solving with the reasoning process of other non-monotonic frameworks may also improve their computational effectiveness.

Abduction and constraint solving has been recently studied by several other works [Fun97], [Kow96], [Den97]. In most of these the integration of the two 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 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.

We have demonstrated, through experiments on the problems of planning and job-shop scheduling, that $ACLP$ offers a high-level representation of the application domain, with the ability to model directly properties of the problem. Furthermore, we have shown that the performance of this approach is comparable to solutions developed directly in CLP languages such as ECLiPSe.

The high-level modeling environment of ACLP offers an extra layer of modularity and flexibility useful for applications with complex and changing requirements. The main dimension of modularity in ACLP applications comes form the fact that we can separate the 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) without affecting the validity of the solution which is ensured by the integrity constraints IC. 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 other major advantage 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 specialized the general model to the needs and preferences of a particular case. Furthermore, the build-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.

References

[Den96] Denecker, M., De Schreye, D.,: SLDNFA: an abductive procedure for abductive logic programs. Journal of Logic Programming, 1996.

[Den97] Denecker, M., Vandecasteele, H., De Schreye, D., Seghers, G., Bayens, T.,: Scheduling by "Abductive Execution" of a Classical Logic Specification. Technical Report, Department of Computer Science, K.U.Leuven, Belgium, 1997.

[ECL94] ECLiPSe User Manual. ECRC, Munich, Germany, 1994.

[Esg88] Esghsi, K.: Abductive Planning with the Event Calculus. Proceedings of the Fifth International Conference on Logic Programming, 1998.

[Esh89] Eshghi, K., Kowalski, R.A.,: Abduction compared with negation by failure. Proceedings of ICLP-89, 1989.

[Gin96] Ginsberg, M.L. : Do Computers Need Common Sense?. Proceedings of the Fifth International Conference on Knowledge Representation, 1996.

[Fun97] Fung, T.F., Kowalski R.A., : The IFF Proof Procedure for Abductive Logic Programming. Journal of Logic Programming, vol. 33(2) pp. 151-165, 1997.

[Kak96] Kakas, A. C.,: Viable Non-monotonic Applications. Proceedings of the ECAI-96 Workshop on "Integrating Non- monotonicity into Automated Reasoning Systems", Budapest, 1996.

[Kak90a] Kakas, A. C., Mancarella, P.,: Database updates through abduction. Proc. 16th International Conference on Very Large Databases, VLDB-90, 1990.

[Kak90b] Kakas, A. C., Mancarella, P.,: On the relation of truth maintenance and abduction. Proc. of the 1st Pacific Rim International Conference on Artificial Intell igence, PRICAI-90, 1990.

[Kak95] Kakas, A.C., Michael, A. : Integrating abductive and constraint logic programming. Procedings of the Twelfth International Conference on Logic Programming, ICLP-95, 1995.

[Kak97] Kakas, A.C., Mourlas, C.: ACLP: Flexible Solutions to Complex Problems. Proceedings of Logic Programming and Non-monotonic Reasoning, 1997.

[Kon90] Konolige, K., : Closure + Minimization Implies Abduction. Proceedings of PRICAI-90, 1990.

[Kow96] Kowalski, R.A., Wetzel, G., Toni, F.,: A Unifying Framework for ALP, CLP and SQO. Technical report, Department of Computing, Imperial College, London, 1996.

[Sad91] Sadeh, N..: Look-Ahead Techniques for Micro-Opportunistic Job Shop Scheduling. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1991.

[Sha97] Shanahan, M.P.,: Event Calculus Planning Revisited. Proceedings of the 1997 European Conference on Planning, ECP-97, 1997.