ACLP: Flexible Solutions to Complex Problems

A.C. Kakas and C. Mourlas

Department of Computer Science, University of Cyprus
75 Kallipoleos str., CY-1678 Nicosia, Cyprus
{antonis, mourlas}@turing.cs.ucy.ac.cy

Abstract. In this paper we present a new system for non-monotonic reasoning performed using abduction. The system, called ACLP, is a programming language based on the framework of Abductive and Constraint Logic Programming (ACLP) which integrates abduction and constraint solving in Logic Programming. It is build on top of the ECLiPSe language for Constraint Logic Programming (CLP) interfacing (and exploiting) appropriately the non-monotonic reasoning of abduction with the specialized constraint solving of the CLP language. ACLP is intended as a programming language that extends the underlying CLP language in which using NMR (in this case abduction) together with constraint solving it is possible to develop flexible solutions that are computationally viable in the real-life domain.

We present the basic theory of ACLP that underlies the system, the main features of the ACLP language and how it can be used when developing applications. We then report on some experiments performed in order to test the cost of the use of the ACLP system as compared with the direct use of the (lower level) constraint solving framework of CLP on which this is build. 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 real-life applications.

1 Introduction

Non-monotonicity enhances the expressive power of a representation framework allowing high-level representations of problems close to their natural specification. The modeling of the problems is thus more direct, modular 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 [1], [2], [3] 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 high-level problem representations and goals to lower level computational problems, of a general problem independent form, whose solutions would solve the original high-level problem goal. It is thus proposed that NMR should be appropriately integrated with specialized constraint solvers for lower-level problem independent domain constraints. Such integrated frameworks will be able to 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 the applications able to respond to their specialized and dynamically changing needs.

In this paper, we present a new system for non-monotonic reasoning performed using abduction. The system, called ACLP, is a programming language based on the framework of Abductive and Constraint Logic Programming (ACLP) [2] which integrates abduction and constraint solving in Logic Programming. It is build on top of the ECLiPSe language [6] for Constraint Logic Programming (CLP) interfacing (and exploiting) appropriately the non-monotonic reasoning of abduction with the specialized constraint solving of the CLP language. The intended use of ACLP system is not to solve computational hard problems of NMR but rather to offer a modeling environment that supports NMR for the development of modular and flexible applications. ACLP is therefore intended as a programming language that extends the underlying CLP language in which using NMR (in this case abduction) together with constraint solving it is possible to develop incrementally flexible solutions that are computationally viable in the real-life domain.

Abductive reasoning has been shown to be appropriate for formulating different application problems in AI e.g. Diagnosis, Planning, Natural Language Understanding and others such as Database Updates. Also abduction has been proved suitable for capturing logically different kinds of inferences such as explanation, non- monotonic and default reasoning, knowledge assimilation and belief revision (see the recent surveys [4, 5]). This versatility of abduction as a reasoning paradigm together with the high level expressivity that it allows are the primary reasons for its success in formulating so many different problems. We therefore take it as given that the ACLP framework and system are appropriate for tackling NMR problems and applications and concentrate here on the issue of the (relative) computational effectiveness of the system.

Two groups of experiments have been performed with ACLP. One to test the cost of the use of the ACLP system as compared with the direct use of the (lower level) constraint solving framework of CLP on which ACLP is build. The other group of experiments aimed to illustrate the high level expressivity of ACLP and the flexibility that it can provide in adapting a program in ACLP 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.

In the next section we present the theoretical foundations of ACLP and its main computational model. In section 3, we briefly discuss its implementation and its main features when used as a programming language for the development of applications. In section 4, we present the experiments carried out to test the viability and usefulness of ACLP.

2 The ACLP framework

The ACLP system is designed as a programming language based on the ACLP framework of integrating Abductive and Constraint Logic Programming [2]. This integration of Abductive Logic Programming (ALP) and Constraint Logic Programming (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, which is satisfiable under a special theory. For the case of ALP this theory is a set of problem specific integrity constraints whereas for the case of CLP this is a built-in problem independent constraint theory. An important observation when integrating these two frameworks is that the interaction between these two types of hypotheses is non-trivial and indeed they can be strongly correlated to each other. For an abducible hypothesis $\exists X ab(X)$ we may also require that the variable(s) X is restricted through some set C of constraints in the constraint domain of CLP. In effect, the hypothesis that we need is $\exists X(ab(X), C(X))$, showing the non-trivial interaction of the two frameworks.

2.1 The Language of ACLP

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 $CLP(\mathcal{R})$ consisting of rules of the form $p_0(t_0) \leftarrow c_1(u_1), \ldots, c_n(u_n) || p_1(t_1), \ldots, p_m(t_m)$[1] where p_i are predicate symbols, c_i are constraints in the domain $\mathcal{R}$ and u_i, t_i are terms of $\mathcal{R}$.
- 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 $CLP(\mathcal{R})$.

A **goal**, G, has the same form as the body of a program rule whose variables are as usual understood as existentially quantified.

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.

[1] Here the symbol $||$ is used to separate the constraint conditions from the program predicate conditions in the conjunction of the body of the rule.

The abducibles are seen as high-level answer holders for goals (or queries) to our program carrying their solutions. **An answer,** Δ, for a goal, G, is 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.

The integrity constraints express high-level 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. In this way, using the integrity constraints we (the user) can express requirements of the problem in an explicit and high-level direct way. More importantly this means that we can separate (isolate) the issue of validity of the solution in the integrity constraints IC from other issues of the problem representation (such as the basic structure of the problem or the quality of the solution) in the program P. This separation can be very useful in the overall development of an application.

2.2 Declarative Non-monotonic Semantics of ACLP

The (non-monotonic) semantics of ACLP is inherited from that of ALP and abduction. An answer for a goal G is correct if it forms an abductive explanation for G. 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 :

- $P \cup \Delta_g$ entails G_g , and
- $P \cup \Delta_g$ satisfies the IC

where G_g denotes a corresponding grounding of the goal G.

Due to lack of space we can not elaborate here on the details of the corresponding grounding and the formal semantics of the integrity constraints (see [2]). Informally, we can consider the integrity constraints as sentences that must be entailed by the program together with the abductive hypotheses ($P \cup \Delta_g$) for Δ_g to be a valid set of hypotheses.

2.3 Computational model of ACLP

A 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 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. The following simple example illustrates the ACLP computation.

Example 1. Consider the following ACLP theory and goal G_0:

$$\begin{aligned} P &= \{p(X) \leftarrow X > 2 \parallel q(X), a(X) \\ &\quad\; q(X) \leftarrow X > 4, X < 10 \parallel []\}, \\ IC &= \{\neg(X > 8 \parallel a(X))\}, \\ G_0 &= p(X), \text{ where ``}a\text{'' is the only abducible predicate.} \end{aligned}$$

In an abductive phase, the initial goal G_0 will resolve against the first and second clauses to obtain the new goal $G_1 = X > 2, X > 4, X < 10 \parallel a(X)$. We then proceed to abduce $\exists X a(X)$ by adding $a(x)$ (x here is a name for this existential variable) 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 the goal $(x > 8 \parallel [])$. This goal must fail. As there are no literals left in the goal, the only way to fail 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.

In this 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.

- check the satisfiability of and reduce (or solve) the constraint store at each step in the abductive phase
- 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 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. From the perpective of the CLP the domain constraints that are needed to solve the high-level goal are generated **dynamicaly** 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.

3 Language Features and Implementation

As descibed in the previous section a program in ACLP has three modules:

- **Module 1:** Contains a standard ECLiPSe program
- **Module 2:** Constains a set of declarations of abducible predicates in the form of ECLiPSe facts as: $abducible\_predicate(predicate\_name/arity)$.
- **Module 3:** constains a set of integrity constraints written as ECLiPSe program facts in the form: $constraint((head :- body))$.

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. Once all three modules have been loaded the program is executed by calling at the ECLiPSe level: *aclp-solve(goal, initial-hypotheses, output-variable)*

- **goal** is an ordinary ECLiPSe goal,
- **initial-hypotheses** is a list of abducible hypotheses, and
- **output-variable** is an ECLiPSe variable.

The *output-variable* returns a list of abducible hypotheses, with their domain variables instantiated to specific values in their domain,containing the *initial-hypotheses* and which is a solution of the goal. Normally, the list of *initial-hypotheses* is empty but this is not necessary as we may want to find solutions to a goal that necessarily contain some hypotheses.

3.1 ACLP Applications

In developing an application with ACLP an important feature of the ACLP language is the fact that we can define for each problem its own problem specific abducibles. These play the important role of answer holders which have higher semantic meaning than simply that of a logical variable (the usual answer holder in LP and CLP). This ability to have high-level answer holders helps in developing high-level and flexible programs as we can use them directly to manipulate the solution and represent explicitly properties of the solution.

Another important feature of the ACLP language is the separation it allows between the program P and the integrity constraints IC. The general problem representation can be divided in two main parts, the basic model of the problem in P and the representation (specification) of what constitutes a valid solution in the integrity constraints IC.

Separating the issue of validity in the constraints from other issues (e.g. quality of a solution) in the model can facilitate the development of applications. It can help in an easier and more modular development especially when we want to make the model more detailed or to change the requirements on the solution. The fact that these two tasks are now decoupled and can be carried out independently is very beneficial in the overall development of an application. It is possible to incrementally refine the model to improve the quality of the solution without affecting its validity (which is always ensured by the integrity constraints in IC). In addition, we can experiment with different design alternatives in the modeling which may improve the quality of the solutions. For example, optimality algorithms can be included in the model of the problem. This gives us the possibility of affecting the search for the solution in order to increase the computational effectiveness of the application system. Moreover, it is possible to exploit natural structures of the application problem in order to build in P a more informed model of the problem that again can result in improvements on the effectiveness of the system. Further details and discussion on this property of separating the issue of validity from that of the optimality of the solution in the context of schedulling applications can be found in [7].

3.2 Implementation

The current version of 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 most difficult task of the meta-interpreter is the management of the two interleaving phases of abduction and consistency during a computation in the ACLP framework. Since constraints can be imposed on 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 (ii) management of the domain variables.

The implementation contains a module for negating the CLP constraints found in the body of an integrity constraint and sending them to the ECLiPSe constraint solver to be satisfied together with the other constraints already set. A binding environment is also constructed during the execution of the meta-interpreter where the domain variable bindings and the constraints attached to these variables are stored. An explicit binding 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.

4 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.

These experiments rest on the premise that ACLP will be computationally viable for real-life applications if its performance is comparable with that of the underlying CLP language or in other words if it does not degrade significantly the performance of the underlying language (which we assume can be used for real-life applications). For this purpose we have selected some standard computationally intensive problems and compared the performance of ACLP with the performance of the underlying language of ECLiPSe on the same problems. We emphasize that these experiments are designed to test explicitly the computational effectiveness of ACLP, in the sense mentioned above, and not its suitability for solving representing NMR application problems such as problems of diagnosis, planning etc which we take as given.

4.1 Increasing the Size of the Problem

The first set of experiments designed to test the computational viability of the proposed system, are based on the *job shop scheduling* problem. In general, we need to schedule 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:

$$\texttt{constraint((:-start(J,T1,R1,S1),T2 is T1}-\texttt{1,start(J,T2,R2,S2),}$$
$$\texttt{duration(T2,D2),S1\#}<\texttt{(S2+D2))).}$$
$$\texttt{constraint((decoupled(T1,S1,T2,S2):-start(J1,T1,R,S1),}$$
$$\texttt{start(J2,T2,R,S2),T1}=\backslash=\texttt{T2)).}$$

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. $decoupled/4$.

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 [8] by putting together two of these problems.

The second set of our experiments is based on the well-known N *queens* puzzle, where we have the requirement to place N queens on a N-*by*-N rectangular board so that no two queens are on the same horizontal, vertical or diagonal line. The ACLP program contains the integrity constraint:

$$\texttt{constraint((:-pos(R1,C1),pos(R2,C2),attack(R1,C1,R2,C2))).}$$

with $pos(Row, Column)$ an abducible predicate and where the definition of $attack$ is the usual one given in the program P. The full ACLP programs can be found in an associated technical report.

The ACLP implementations of these problems were tested against corresponding implementations directly in ECLiPSe.

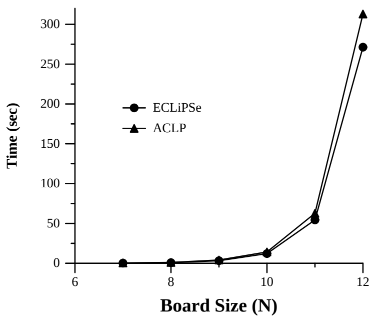

Fig. 1. *All solutions, N-queens problem using ECLiPSe and ACLP*

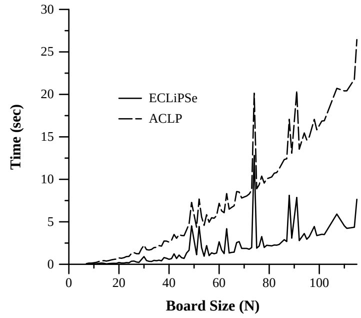

Fig. 2. *One solution, N-queens problem using ECLiPSe and ACLP*

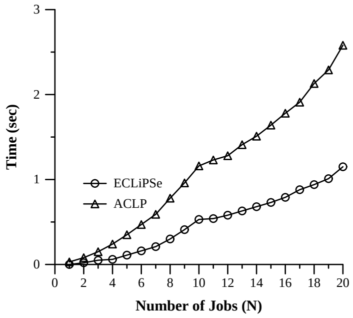

Fig. 3. *One solution, job shop scheduling problem using ECLiPSe and ACLP*

Figure 3 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. As we will also see in the next subsection the constraints are expressed in a form closely to their natural specification in the ACLP programs whereas more effort was necessary for encoding the same constraints directly in ECLiPSe.

Similar results were taken from the execution of the N-queens problem, where only one solution for the placement of the queens was required from both the implementations. We compared the execution times for different board sizes ranging from 7 to 115. Figure 2 presents the results where we can see that ACLP is still performing with times comparable to the ones achieved by the ECLiPSe implementation. This different is a constant factor of 4 to 5 times faster for ECLiPSe.

Finally, Figure 1 displays the run-time costs of the N-queens problem, where all solutions are required for different board sizes ranging from seven to twelve. The performance of the ACLP, as it can be observed follows closely the performance of the ECLiPSe implementation.

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 its 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: 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 (1) (for the specific case of $\tau_0 = 14$ and $t_0 = 21$)

$$\texttt{constraint}((: -\texttt{start}(1,14,\texttt{R0},\texttt{S0}), \texttt{start}(\texttt{J},\texttt{T},\texttt{R},\texttt{S}), \texttt{T} = \backslash = 14, \\ \texttt{S0}\# > 21, \texttt{S}\# > \texttt{S0})). \qquad (1)$$

This constraint was implemented also in the ECLiPSe language. The performance of each approach is presented in Table 1 on the second column named $Constr1$, 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 computation 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 (2), (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*2 of the Table 1.

$$\texttt{constraint((:-start(J,T,1,S),start(2,23,1,Sd),duration(23,Dd),}$$
$$\texttt{S\# > Sd,S\# < Sd + Dd + 20)).} \quad (2)$$

We must point out that in both these cases of adding the constraints (1) and (2) 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 (3), 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*3 of Table 1. Due to the much needed effort to implement this directly in ECLiPSe we did not carry this out!

$$\texttt{constraint((:-start(J,T,0,S),start(J1,T1,0,S1),start(J2,T2,0,S2),}$$
$$\texttt{T1 = \textbackslash = T2,0\# < S1,S1\# < 0 + 32,}$$
$$\texttt{0\# < S2,S2\# < 0 + 32,S\# > 32,S\# < 32 + 10.)).} \quad (3)$$

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 (4) and the performance of its implementation is presented in Table 1. Again this constraint was not implemented directly in ECLiPSe.

$$\texttt{constraint((:-start(J1,T1,R1,S1),start(J2,T2,R2,S2),T1 = \textbackslash = T2,}$$
$$\texttt{related(R2,R1),duration(T2,D),S2\# < S1,S1\# <= S2 + D)).} \quad (4)$$

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 (5):

$$\texttt{constraint((working(R2,S):-start(J,T,R1,S),related(R2,R1))).} \quad (5)$$

where `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.
```

Performance Measurements (in secs)						
System	Standard Configuration	$Constr1$	$Constr2$	$Constr3$	$Constr4$	$Constr5$
$ACLP$	0.90	1.04	0.94	1.34	3.45	1.62
ECL^iPS^e	0.42	0.68	0.42	*	*	*

Table 1. Performance Measurements as the Complexity of the Problem Increases

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 1.

Our conclusions based on these results can be summarized as follows:

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 spend 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 for problems with changing specifications.

We note that as with ECLiPSe and other CLP languages, the performance of the ACLP system can sometimes be sensitive to the order in which the constraints are written (and hence to the order which the meta-interpreter considers these constraints).

Finally, we also mention that a simpler implementation of ACLP that does not exploit fully the integration of NMR and constraint solving has already been used to develop two real-life applications: a university timetabling problem and an airline crew-rostering problem [7].

References

1. Ginsberg, M.L. : Do Computers Need Common Sense?. Proceedings of the Fifth International Conference on Knowledge Representation, 1996.
2. Kakas, A.C., Michael, A.. : Integrating abductive and constraint logic programming. Procedings of the Twelfth International Conference on Logic Programming, Tokyo 1995.
3. Kakas, A.C.: Viable Non-monotonic Applications. Proceedings of the ECAI-96 Workshop on "Integrating Non- monotonicity into Automated Reasoning Systems", Budapest, 1996.
4. Kakas, A.C., Kowalski, R.A., Toni, F. : Abductive Logic Programming., Journal of Logic and Computation, 2(6), 719-770, 1993.
5. Paul, G. : Approaches to abductive reasoning: an overview. Artificial Intelligence Review, 7, 109-152, 1993.
6. ECLiPSe User Manual. ECRC, Munich, Germany, 1994.
7. Kakas, A.C., Michael, A.. : Applications of Abductive Logic Programming: a case for Non- monotonic reasoning. University of Cyprus Technical report, TR-97-3, 1997.
8. Sadeh, N..: Look-Ahead Techniques for Micro-Opportunistic Job Shop Scheduling. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1991.