Two Problems - Two Solutions: One System - ECLiPSe*

Mark Wallace and André Veron

April 1993

1 Introduction

The constraint logic programming system ECL^{i}PS^e [4] is the successor to the CHIP system [1]. Computation in ECL^{i}PS^e alternates between two modes: constraint handling and host program execution. The host programming language is (an extended) Prolog, which handles search, and interaction with the programming environment. The control for host program execution mode is the usual Prolog control. The constraint handling mode has a quite different form of control, which generalises data-driven computation. During constraint handling all possible information is extracted from the constraints. When there is no more information to be extracted, the system returns to host program execution, which continues until another constraint is posted and constraint handling restarts.

In the next section we show the advantage of combining Prolog programming with constraints handling for a shift planning application. In the third section we indicate how to control the constraint handling itself, and the application of such control for optimising job-shop scheduling programs.

<u>Shift Planning Application</u> The problem is to decide the best shift rota to make optimal use of a resource which is only available for a restricted time each day. Clearly there is a limit on the weekly hours for each shift worker, and there are other application-specific constraints. The problem is to choose start times and end times for the shifts throughout the week.

The difficulty lies in the cost function, since the hourly pay varies with the time of day and day of the week. It quickly emerged that the best approach is to divide the week into half hour periods, and associate finite domains with the start and end times of the shifts. Unfortunately the constraints do not prune the search space sufficiently to allow the optimum to be found in a practical time. The reason is that the problem exhibits a huge number of symmetries

*This paper appears in the Proceedings of the IEE Colloquium on Advanced Software Technologies for Scheduling, London, April 1993

(based on extending one shift by a period of time, and reducing another by the same period).

A solution was finally reached using an approach based on [3]. The approach is an extension of backchecking, and relies on the concept of a *nogood* environment in the ATMS. We now show how *nogoods* were programmed in ECL^{i}PS^e for this application.

The search routine is an essential component of most CLP programs, and in the shift planning application it appears as a *labelling* procedure. This procedure, which non-deterministically assigns values to the problem variables, has the form

```
label(Vars) :- Vars=[].
label(Vars) :- Vars=[V|Rest],
    chooseval(V),
    label(Rest). % Recursive call to label routine
```

Suppose the procedure `chooseval` which assigns a value to a variable makes some bad choices, and at some point all attempts to label the remaining variables fail. If $V1 = Val1, ..., Vn = Valn$ are the bad choices already made when backtracking occurs, the system has proved that there is no solution with these values for these variables. However there is little use in recording this deduction, since the system will never try the same partial labelling again.

However taking the problem constraints into account, we can draw a more general conclusion about the cause of failure. As a trivial example, suppose the problem has just one constraint, that the sum of all the variable values must be greater than some given constant (i.e. $V1 + V2 + ... + Vm > Const$, where m is the number of problem variables). Now when backtracking occurs as outlined above, we can draw a much more useful conclusion than just saying $V1 = Val1, ..., Vn = Valn$ is no good. We can conclude that, whatever values are chosen for $V1, ..., Vn$, if $V1 + V2 + ... + Vn =< Val1 + Val2 + ... + Valn$ then this partial labelling will still be no good!

ECRC, Arabella Strasse 17, 8000 Munich 81, Germany

For the shift planning problem we introduced just such a measure. If a partial assignment of start and end times to shifts leads to a failure, the program calculates the cost Cn of the partial assignment, and the period of time Un during which the resource is unused. If any other partial assignment leads to a cost $C >= Cn$ and unused time $U =< Un$, then this will lead either to a failure or to a solution with greater cost than one already found.

The extension of the labelling routine to check for *nogood* is simple in ECL^{i}PS^e. First extra parameters are introduced to record the cost and unused times for the current partial labelling. Then an extra clause is added to check if the current partial assignment is no good. If it is no good, the subgoals

`!,fail` cause the current labelling goal to fail too. Finally a last clause is added to update the *nogood*s when backtracking occurs:

```
label(Vars,Cost,Unused) :- Vars=[]. % Unchanged
label(Vars,Cost,Unused) :-
    recorded(nogood,[Cn,Un]),       % Find a(nother) nogood
    Cost >= Cn, Unused =< Un,       % Test it
    !, fail.                        % Fail the labelling call
label(Vars,Cost,Unused) :- Vars=[V|Rest],
    chooseval(V),
    calc(NCost,NUnused,Cost,Unused,V),  % Calc new cost and unused
    label(Rest,NCost,NUnused).          % Use new cost and unused
label(Vars,Cost,Unused) :-
    record(nogood,[Cost,Unused]),   % Record another nogood
    fail.                           % Fail the labelling clause
```

Not only is the extension quite trivial in ECL^{i}PS^e, it also leaves the usual constraint handling unaffected. The resulting program therefore uses reactive constraints to prune the search tree and *nogood*s as an additional mechanism. Experiments showed that neither feature alone enabled a solution to be found - it was the combination which made the problem tractable to ECL^{i}PS^e.

2 Constraints Handling for Job Shop Scheduling

In this section we concentrate on constraint handling. We distinguish two varieties of constraints: primitive constraints and reactive constraints. Primitive constraints are added, during computation, to a global constraint store. Thus primitive constraints are a generalised form of data. They allow not only a value to be associated with a variable (as in traditional data stores) but any primitive constraint to be imposed on any variable or variables. To ensure that the system does not hold contradictory constraints in its store, such as $X > Y$ and $Y > X$, ECL^{i}PS^e provides a global procedure which to decide the consistency of each new primitive constraint with the current store. The CLP Scheme [2] describes the integration of primitive constraints in logic programming.

Different classes of primitive constraints admit different decision procedures. For example the decision procedure for linear equations and inequations over the rationals can be used to detect the inconsistency of the above constraints. However such a decision procedure can be computationally expensive when the number of primitive constraints in the store grows.

We now introduce reactive constraints. Reactive constraints are agents with a behaviour which may be either program-defined or built into ECL^{i}PS^e. Shortly, the behaviour is as follows: the agent does nothing until a certain condition (or *guard*) is entailed by the current constraint store; then it transforms itself into a new set of agents, possibly adding new primitive constraints to the store. Often a reactive constraint is defined by a (finite) number of alternative possible behaviours. A behaviour is then chosen non-deterministically at runtime.

A very simple example of a reactive constraint which occurs in every scheduling problem is the constraint that two tasks cannot run on a machine at the same time. If $S1$ and $D1$ are the start time and duration of one task, and $S2$ and $D2$ are those for another, the reactive constraint can be defined as follows:

```
nonOverlap(S1,D1,S2,D2) ==> S1+D1>S2 | S1 >= S2+D2
nonOverlap(S1,D1,S2,D2) ==> S2+D2>S1 | S2 >= S1+D1
```

This constraint has two alternative behaviours, one specified by each line in the definition. Informally this definition makes sure that the two tasks do not overlap by adding, as soon as one task A cannot be completed before the other B starts, a constraint that A must start after B is completed.

The first line says that if the current constraint store entails the guard `S1+D1>S2` then the reactive constraint `nonOverlap(S1,D1,S2,D2)` transforms itself into the empty set of constraints (i.e. it disappears), and a new primitive constraint $S1 \geq S2 + D2$ is added to the constraint store. The second line defines a symmetric behaviour. The `nonOverlap` constraint can be used for solving small scheduling problems, but it generates a large number of primitive constraints and the linear solver mentioned above quickly becomes too inefficient for tackling non-toy scheduling problems.

An advantage of reactive constraints is that their behaviour is program defined. Thus we can define a reactive behaviour for constraints that is weaker, and computationally less costly, than solving. A simple example is the treatment of non-linear equations by delaying them until they are linear. A more radical example is the treatment of linear inequations and disequations by finite domain propagation as in cc(FD) [5]. This is (usually) much cheaper than treating them as primitive constraints and testing them for global satisfiability. Propagation is simply a reactive constraint behaviour.

Using this approach we can replace the primitive constraints `S1 >= S2+D2` and `S2 >= S1+D1` in the definition of `nonOverlap` with reactive constraints which perform finite domain propagation. The resulting scheduling program is efficient enough to solve small, but non-toy, problems.

For larger more complex scheduling problems it is necessary to perform more global reasoning, on all the tasks which run on a single machine, rather than just pairs of such tasks. Such global reasoning is also straightforward using reactive constraints, though the form of the guards required is more general. For example we need to extract the earliest possible start time and the latest possible end time for a set of tasks, and the sum of their durations. ECL^{i}PS^e provides facilities for accessing this information, and for tailoring constraint propagation to the application at hand.

Explicit control of constraint behaviour has been used to obtain a guaranteed optimal solution the 10X10 job shop problem. More generally, the integration of primitive constraint solving, reactive constraint behaviours and host program control provide a complete armoury for tackling practical job shop scheduling problems.

3 Conclusion

With ECL^{i}PS^e we have introduced a second generation constraint logic programming language in which the advantages of CHIPs built-in constraint handling have been retained while "opening" the box and enabling users to construct new reactive constraints. Moreover the open architecture of ECL^{i}PS^e has enabled us to add new solvers and test new features, such as intelligent backtracking, without modifying the system itself. Ongoing experimentation with practical problems continues to reveal the efficiency and expressive power of ECL^{i}PS^e .

References

[1] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, December 1988.

[2] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages (POPL'87)*, Munich, FRG, January 1987.

[3] F. Maruyama, Y. Minoda, Sawada S., and Y. Takizawa. Constraint satisfaction and optimisation using nogood justifications. In *Proc. 2nd Pacific Rim Conf. on AI*, 1992.

[4] M. Meier, J. Schimpf, and et.al. ECL^{i}PS^e , ecrc common logic programming system, user manual. Technical Report TTI/3/93, ECRC, 1993.

[5] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58, 1992.