Towards a declarative and efficient glass-box CLP language

Robert A. Kowalski Francesca Toni Gerhard Wetzel
Department of Computing
Imperial College, London SW7 2BZ, UK
{rak,ft,gw1}@doc.ic.ac.uk

Introduction

This short paper is a preliminary report on ongoing research and will focus on motivating a new direction for Constraint Logic Programming (CLP). Using a non-trivial example, it will illustrate the advantages of the proposed *glass-box* CLP language, i.e. a language which allows the user to specify how a program should handle and solve constraints instead of relying on a hidden constraint solver, a *black-box*, to do this. In the language we suggest (see also [Ko92,Fu93,To94,We94]), knowledge about constraints is expressed by *forward propagation rules* (FPRs) of the form

$$A_1 \wedge \ldots \wedge A_m \Rightarrow B_1 \wedge \ldots \wedge B_n$$

where the A_i are conditions ensuring that the conclusions $B_1, \ldots, B_n$ hold. Disjunctions are allowed in goals, and FPRs may be used to propagate $false$ into a disjunct, which can then be eliminated.

Glass-box versus black-box approach

CLP languages like CHIP ([VH89]) or other instances of the CLP(X) language family ([JaLa87]) allow the user to write Prolog-style clauses containing (often only predefined) constraints, which are then handed to a constraint solver – a black-box from the user's point of view. The user just states the constraints to define a problem, while the constraint solver takes care of finding a solution, using the methods supplied by its designers rather than by its users. Many examples are given in [VH89] for which this appears to be an acceptable approach, but there are other problems for which the black-box approach seems to be unsatisfactory, where detailed problem specific knowledge guiding the way to a solution has to be incorporated, i.e. where a glass-box approach is needed (see [Fr92] and [CaLa94] for similar arguments and examples). One such example, the job-shop scheduling problem, is discussed in the next section.

Job-shop scheduling as an example application

In a job-shop scheduling problem (JSSP) m machines and n jobs are given. A job consists of a number of ordered tasks, each of which has to be executed on one of the m machines. A job must not use the same machine twice for different tasks, and a machine can only process one task at a time. The goal is to minimise the *makespan*, which is defined as the time by which all jobs have been completely processed.

For a task T_i, let $r(i)$ be a variable denoting its actual starting time ("release date"), s_i be its currently known earliest possible starting time, f_i be its currently known latest possible finishing time, and p_i be its given processing time. There are then four classes of constraints in a JSSP:

1. $r(j) \geq r(i) + p_i$, if T_i and T_j are tasks in the same job and T_j has to be processed after T_i.
2. $(r(i) \geq r(j) + p_j) \ \vee \ (r(j) \geq r(i) + p_i)$, if T_i and T_j are tasks in different jobs which have to be processed on the same machine.
3. $s_i \leq r(i)$ for all tasks T_i.
4. $r(i) + p_i \leq f_i$ for all tasks T_i.

A solution to a JSSP is an assignment of values to every $r(i)$ such that 1.–4. are satisfied. It is the disjunctive nature of the second class of constraints which makes JSS an intractable and in fact $\mathcal{NP}$-complete problem. Due to the complexity of the problem, simply stating all the constraints and handing them to an all-purpose theorem-prover with a built-in constraint solver is not feasible — unless the constraint solver has been adequately prepared. Whereas in a black-box approach this requires changing the compiler (or interpreter), a glass-box approach allows the user to incorporate methods into the program, which reduce the number of splitting steps required before a solution is generated, and such methods may include algorithms from Operations Research (OR).

The glass-box approach we propose is based on a theorem-proving framework where goals assume the form of conjunctions of disjunctions. In the scheduling application, a part of an intermediate goal may look like this:

$$[(r(2) \geq r(1)+2) \ \vee \ (r(1) \geq r(2)+3)] \ \wedge \ (r(2) \geq 2) \ \wedge \ (3 \geq r(1)) \ \wedge \ \ldots$$

and the problem-solver is designed to replace goals by equivalent goals, replacing disjuncts by $false$ whenever possible. In particular, goals can be unfolded by means of if-and-only-if definitions and logically redundant information can be added by propagating with forward propagation rules (FPRs).

In the scheduling application, if-and-only-if definitions of the form

$$\texttt{ordered(I,J)} \leftrightarrow \ \exists\, \texttt{P} \ (\ \texttt{proctime(J,P)} \wedge \texttt{r(I)} \geq \texttt{r(J)+P} \ ) \ \vee$$
$$\exists\, \texttt{P} \ (\ \texttt{proctime(I,P)} \wedge \texttt{r(J)} \geq \texttt{r(I)+P} \ )$$

can be used to derive goals of the form above from goals of the form $\texttt{ordered(1,2)} \wedge \texttt{(r(2)} \geq \texttt{2)} \wedge \texttt{(3} \geq \texttt{r(1))}$. Moreover, properties of $\geq$ like transitivity and

$$X \geq Y \ \wedge \ Y > X \ \Rightarrow \ false$$

can be used as FPRs. These rules can be applied to add conjuncts inside single disjuncts in the goals by using information local to the disjuncts as well as information global to all disjunctions. In particular, the addition of $false$ to a disjunct is logically equivalent to eliminating it. By applying this propagation strategy to the goal above, $r(1) \geq 5$ first and then $false$ are added to the second disjunct by propagating with the FPRs for $\geq$ from $r(1) \geq r(2)+3 \ \wedge \ (r(2) \geq 2) \ \wedge \ (3 \geq r(1))$. Therefore, such a disjunct can be eliminated, and splitting (i.e. distribution of conjunction over disjunction) is not required. The same strategy, to eliminate disjuncts before splitting, is at the core of efficient OR approaches to a JSSP.

FPRs are similar to the *constraint handling rules* (CHRs) proposed in [Fr92]. However, while in our approach FPRs are combined with if-and-only-if definitions, CHRs are embedded in a given host language such as Prolog. Moreover, while in our framework goals are conjunctions of disjunctions, in [Fr92] goals are conjunctions of atoms. As a consequence, it is not obvious how to simulate in the language suggested in [Fr92] the use of FPRs illustrated above, of eliminating disjuncts by propagating global information locally to a disjunct.

For the scheduling application, we can obtain other FPRs by analysing algorithms used in OR. In the rest of the paper, we will analyse the OR techniques proposed in [CaLa94], based on algorithms described in [ApCo91] and the literature referenced therein. [CaLa94] describes an efficient program to solve a JSSP in a hybrid (declarative and procedural) language called LAURE. For obvious reasons, such as verifying programs relative to specifications, it is desirable to eliminate the non-logical elements of the LAURE program, e.g. destructive assignment, without losing too much efficiency; the framework proposed here might be a way to achieve this.

Logically, what makes the LAURE program efficient is its use of *task intervals*. A task interval $[T_i, T_j]$ is defined for every pair of tasks (T_i, T_j) which have to be processed on the same machine (note that $T_i = T_j$ is allowed) as the set of all tasks T_k on the same machine whose earliest possible starting time s_k is greater than or equal to that of T_i and whose latest possible finishing time f_k is smaller than or equal to that of T_j:

$$[T_i, T_j] = \{T_k : s_k \geq s_i \wedge f_k \leq f_j\}$$

A task interval may be empty and is then *inactive*. The contents of each of the mn^2 task intervals are stored together with the sum $p_{[T_i,T_j]}$ of the processing times of all its tasks in a matrix which gives immediate access to an interval via either of its two defining tasks. This direct access is crucial when changes to the lower and upper bounds s_k and f_k are propagated by propagation rules which are very similar in form to FPRs. For example, the *edge finding* rule says that a task cannot be scheduled first among the tasks in a task interval, if this makes the sum of the processing times in the interval greater than the gap between the latest possible finishing and the earliest possible starting time:

$$T_k \in [T_i, T_j] \ \wedge \ p_{[T_i,T_j]} > f_j - s_k \ \Rightarrow \ s_k \geq min\{s_l + p_l : T_l \in [T_i, T_j], l \neq k\}$$

Transforming this rule into a FPR is easy:

$\texttt{task\_interval(T}_i\texttt{, T}_j\texttt{, Tasks, PSum)} \wedge \texttt{member(T}_k\texttt{, Tasks)} \wedge \texttt{proctime(T}_j\texttt{, P}_j\texttt{)} \wedge$

$\texttt{r(T}_j\texttt{)+P}_j \leq \texttt{F}_j \wedge \texttt{r(T}_k\texttt{)} \geq \texttt{S}_k \wedge \texttt{PSum} > \texttt{F}_j\texttt{-S}_k \wedge \texttt{findmin(Tasks,T}_k\texttt{,NewS)} \Rightarrow \texttt{r(T}_k\texttt{)} \geq \texttt{NewS}$

If applicable, the FPR generates a new lower bound for $r(T_k)$. This can, but need not be tighter than the old one, but by employing transitivity (possibly using further FPRs, cf. [To94]) only the tighter one is kept, although it will be useful to store the old value in, say, $\texttt{old\_s(T}_k\texttt{, S}_k\texttt{)}$ to access it again later (see below).

If the new lower bound for $r(T_k)$ is tighter than the old one, some task intervals on the considered machine may change. There are four different types of change: deactivating intervals, removing a task from an interval, creating new active intervals, and adding a task to an interval. For example, an interval $[T_i, T_j]$ is deactivated if the earliest possible starting time of T_i is strictly greater than the latest possible finishing time of T_j. All these changes can again be expressed in terms of FPRs, e.g. here is the one to deactivate an interval:

$\texttt{old\_s(T}_i\texttt{, OldS}_i\texttt{)} \wedge \texttt{r(T}_i\texttt{)} \geq \texttt{NewS}_i \wedge \texttt{OldS}_i \texttt{<NewS}_i \wedge \texttt{task\_interval(T}_i\texttt{, T}_j\texttt{, Tasks, PSum)} \wedge$

$\texttt{proctime(T}_j\texttt{, P}_j\texttt{)} \wedge \texttt{r(T}_j\texttt{)+P}_j \leq \texttt{F}_j \wedge \texttt{NewS}_i \texttt{ > F}_j \Rightarrow \texttt{task\_interval(T}_i\texttt{, T}_j\texttt{, [], PSum)}$

The formulation we suggest gives direct access to all stored data, i.e. to the task intervals and to the earliest possible starting and latest possible finishing times. In the version of [CaLa94], all of these are updated by means of destructive assignment. It might be argued that the rule above can be implemented in a manner which simulates destructive assignment by making use of an optimisation similar to tail recursion. Such an optimisation can be justified by the connection graph procedure which eliminates used links and pure clauses (containing a literal without any link), e.g. $\texttt{task\_interval(T}_i\texttt{, T}_j\texttt{, Tasks, P)}$ can be replaced by $\texttt{task\_interval(T}_i\texttt{, T}_j\texttt{, [], P)}$ and thus deactivated because $\texttt{task\_interval(T}_i\texttt{, T}_j\texttt{, Tasks, P)}$ is no longer linked to any other clauses in the program. To obtain the advantage of direct accessibility, a Prolog program or a Prolog-based CLP implementation could store the data as a database of assertions and update them using `assert` and `retract`.

Related work and conclusion

Arguments against the black-box CLP approach have in part motivated the cc(FD) language described in [VHSaDe93]. cc(FD) allows more user-defined control over the search strategy; however, it still relies on a fixed built-in constraint solver. A cc(FD) program solved a 10 machines, 10 jobs JSS benchmark in 90 hours, which does not compare well with the 7 minutes needed by the OR algorithms of [ApCo91], which have been implemented in C. It is not clear how the cc(FD) approach could incorporate OR techniques of the kind discussed in this paper to improve its efficiency.

[CaLa94] describes an efficient program to solve a JSSP, whose performance is comparable with those of OR algorithms. For example, it solves the 10×10 JSS benchmark in about 20 minutes. This program is implemented in a hybrid (procedural and declarative) glass-box CLP language. In this paper we have illustrated how it might be possible to encode the techniques suggested in [CaLa94] in a purely declarative glass-box CLP language.

FPRs are similar both in syntax and semantics to the CHRs defined in [Fr92], but FPRs are used in the context of if-and-only-if definitions, while CHRs are used with Prolog-style programs. The explicit representation of disjunctions in our approach is important, because it enables us to reduce disjuncts to $false$ without splitting. This strategy seems to be built into many OR algorithms. It is not clear how Frühwirth's CHR approach could achieve a similar behaviour without the implementation of a meta-interpreter.

To conclude, although there is already a rather large number of (C)LP languages, an efficient glass-box language which is purely declarative, yet allows a procedural reading of its propagation rules seems to have a great potential.

Acknowledgements

The first two authors were supported by the Fujitsu Research Laboratories and are grateful to Ken Satoh and Fumihiro Maruyama for many helpful discussions.

References

[ApCo91] Applegate, D.; Cook, W.: A computational study of the job-shop scheduling problem, ORSA Journal on Computing 3 (1991) No. 2

[CaLa94] Caseau, Y.; Laburthe, F.: Improved CLP Scheduling with Task Intervals, Proc. of the 11^{th} ICLP, pp. 369-383, MIT Press 1994

[Fr92] Frühwirth, T.: Constraint Simplification Rules, ECRC Technical Report 92-18, 1992

[Fu93] Fung, T. H.: Theorem proving approach with constraint handling and its applications on databases, MSc Thesis, Imperial College, London 1993

[JaLa87] Jaffar, J.; Lassez, J.-L.: Constraint Logic Programming, Proc. of the 14^{th} ACM Symp. on the POPL 1987, pp. 111-119

[Ko92] Kowalski, R. A.: A dual form of logic programming. Lecture Notes, Workshop in Honour of Jack Minker, University of Maryland, November 1992

[To94] Toni, F.: A theorem-proving approach to job-shop scheduling, Imperial College 1994

[VH89] Van Hentenryck, P.: Constraint Satisfaction in Logic Programming, MIT Press 1989

[VHSaDe93] Van Hentenryck, P.; Saraswat, V.; Deville, Y.: Design, Implementation, and Evaluation of the Constraint Language cc(FD), Brown University Technical Report No. CS-93-02, 1993

[We94] Wetzel, G.: Scheduling in a New Constraint Logic Programming Framework, MSc Thesis, Imperial College 1994