ECAI 2008
M. Ghallab et al. (Eds.)
IOS Press, 2008

doi:10.3233/978-1-58603-891-5-903

Integrating Abduction and Constraint Optimization in Constraint Handling Rules

Marco Gavanelli and **Marco Alberti** and **Evelina Lamma** [1]

1 Abduction and CHR

Abductive Logic Programming (ALP) [10] is a set of languages supporting hypothetical reasoning; the corresponding proof-procedures feature a simple, sound implementation of negation by failure [6]. An ALP is a logic program KB with a distinguished set $\mathcal{A}$ of predicates, called *abducibles*, that do not have a definition, but their truth value can be *assumed*. A set of implications called *Integrity Constraints* (IC) restrict the possible set of hypotheses, in order to avoid unrealistic assumptions. Given a goal $\mathcal{G}$, the aim is to find a set $\Delta \subseteq \mathcal{A}$ such that $KB \cup \Delta \models \mathcal{G}$ and $KB \cup \Delta \models \mathcal{IC}$.

ALP and Constraint Logic Programming (CLP) have been merged in works by various authors [11, 12, 5]. However, while almost all CLP languages provide algorithms for finding an optimal solution with respect to some objective function (and not just *any* solution), the issue has received little attention in ALP. We believe that adding optimisation meta-predicates to abductive proof-procedures would improve research and practical applications of abductive reasoning.

However, abductive proof-procedures are often implemented as Prolog meta-interpreters, which makes clumsy the strong intertwining with CLP required to fully exploit optimisation meta-predicates.

In the line with previous research [1, 9, 4, 2], we implemented the $\mathcal{S}$CIFF abductive proof-procedure [3] in Constraint Handling Rules (CHR) [7], which provides a strong integration between abduction and constraint solving/optimisation. In $\mathcal{S}$CIFF, the abductive logic program can invoke optimisation meta-predicates, which can invoke abductive predicates, in a recursive way.

Previous implementations of abduction in CHR mapped abducibles into CHR constraints, and integrity constraints into CHR rules [1, 9, 4, 2]. In this way, the implementation is very efficient, but there are limitations on the language: only abducibles can occur in the condition of ICs. This limits the applicability of sound negation by failure to abducibles, while negative literals of other predicates inherit *"the dubious semantics of Prolog"* [4].

E.g., the following IC (where abducibles are in **bold**)

$$\mathbf{a}(X,Y), \mathbf{b}(Y) \rightarrow \mathbf{c}(X) \wedge p(Y) \vee q(X) \tag{1}$$

can be rewritten as a propagation CHR

$$\texttt{a(X,Y)}, \texttt{b(Y)} \texttt{ ==> } \texttt{c(X)}, \texttt{p(Y)} \texttt{ ; } \texttt{q(X)}$$

because in the antecedent only abducibles occur, thus in the head of the propagation CHR there are only CHR constraints. Instead, the IC

$$\mathbf{a}(X,Y), p(Y) \rightarrow r(X) \wedge q(Y) \vee q(X), \tag{2}$$

cannot be represented in this way, because $p/1$ is not abducible. This means that it is not possible to deal with negation by failure in a sound way, since $not(p(X))$ should be rewritten as $p(X) \rightarrow false$.

In $\mathcal{S}$CIFF, an abducible $\mathbf{a}(X,Y)$ is represented as the CHR constraint $\texttt{abd(a(X,Y))}$. We do not map integrity constraints to CHR rules, but to other CHR constraints. IC (2) is mapped to the constraint

$$\texttt{ic}([\texttt{abd(a(X,Y))}, \texttt{p(Y)}], [[\texttt{r(X)}, \texttt{q(Y)}], [\texttt{q(X)}]]).$$

The operational semantics (derived from the IFF [8]) is defined by a set of transitions [3]. The transitions are then easily implemented as CHR rules; for example, transition *propagation* (joined with *case analysis*) [8] propagates an abducible with an implication:

$$\texttt{abd(P)}, \texttt{ic}([\texttt{P1}|\texttt{Rest}], \texttt{Head}) \Longrightarrow$$
$$\texttt{rename}(\texttt{ic}([\texttt{P1}|\texttt{Rest}], \texttt{Head}), \texttt{ic}([\texttt{RenP1}|\texttt{RenRest}], \texttt{RenHead})),$$
$$\texttt{reif\_unify}(\texttt{RenP1}, \texttt{P}, \texttt{B}),\ (\texttt{B} = 1, \texttt{ic}(\texttt{Rest}, \texttt{Head}); \texttt{B} = 0)$$

We first `rename` the variables (considering their quantification), and then apply *reified unification* [12]: a CHR constraint that imposes that either the two first arguments unify and $B = 1$, or that the two arguments do not unify and $B = 0$.

One of the features of the CHR implementation is that the abductive program written by the user is directly executed by the Prolog engine, and the resolvent of the proof-procedure coincides with the Prolog resolvent. This also means that every Prolog predicate can be invoked, and, in particular, we can invoke optimisation meta-predicates: in some cases, it is not enough to find *one* abductive solution, but the *best* solution with respect to some criteria is requested. CLP offers an answer to this practical need by optimisation meta-predicates (`minimize` and `maximize`), that select the best solution amongst those provided by a goal.

2 An example from Game Theory

N grim pirates plundered a treasure of M golden coins. They have to divide their treasure, and they want to have fun. Since they are bloodthirsty, they adopt rules in which blood might be shed:

1. The lowest pirate in grade proposes a full division: he decides how many coins are given to each pirate (including himself).
2. All the pirates vote: if the majority votes for the proposal, the money is shared as in the division. Otherwise, the proposer is killed, and the process restarts from step 1.

Knowing that all pirates are greedy and bloodthirsty (i.e., they mostly care about money, and in case of parity they like to see someone die), we have to propose a division.

[1] University of Ferrara, Italy, email: name.surname@unife.it

This is clearly an optimisation problem, as pirates want to get as much money as possible; moreover, the proposer has to hypothesise how the other pirates will vote, in order to stay alive.

The lowest in grade will abduce an atom bearing the information for each pirate: at the first proposal, there is a literal for each pirate

$$\mathbf{E}(pirate(Grade, Vote, Coins, Alive), 1) \tag{3}$$

meaning that the proposer gives to the pirate with given $Grade$ (1 being the highest) a number $Coins$ of coins, we suppose his vote is expressed with a boolean $Vote$ (an integer 0=*false* or 1=*true*), and that at the end he will be alive if and only if the boolean $Alive = 1$.

Moreover, we had better try to foresee what could possibly happen in the next protocol iterations, in the unlucky case our proposal does not get the majority. We suppose each proposal happens at a time step indicated by an integer (last argument of Eq. 3).

Now we can see the rules of the protocol. Predicate `npirates/1` defines the number of pirates. The N-th pirate makes the first proposal, the $N-1$ has the second choice and so on:

$$turn(Grade, Turn) \text{:-} npirates(N), T = N + 1 - Grade.$$

Each pirate is alive, if his turn of proposing has not come yet.

$$\mathbf{E}(pirate(Grade, Vote, Coins, Alive), T) \\ \wedge turn(Grade, Turn) \wedge T < Turn \rightarrow Alive = 1$$

After his proposal, a pirate is dead: he gets 0 coins and does not vote:

$$\mathbf{E}(pirate(Grade, Vote, Coins, Alive), T) \\ \wedge \quad turn(Grade, Turn) \wedge T > Turn \\ \rightarrow \quad Alive = 0 \wedge Vote = 0 \wedge Coins = 0$$

Each pirate votes for his own proposal:

$$\mathbf{E}(pirate(Grade, Vote, Coins, Alive), T) \\ \wedge turn(Grade, Turn) \wedge T = Turn \rightarrow Vote = 1$$

If in the current proposal I suppose to get more money than in the next, I will vote for the current one. Otherwise, I will vote against: either I hope to get more money, or I hope to see the proposer die.

$$\mathbf{E}(pirate(Grade, Vote_1, Coin_1, Alive_1), T_1) \wedge T_2 = T_1 + 1 \wedge \\ \mathbf{E}(pirate(Grade, Vote_2, Coin_2, Alive_2), T_2) \rightarrow \\ Coin_1 > Coin_2 \wedge Vote_1 = 1 \vee Coin_1 \leq Coin_2 \wedge Vote_1 = 0.$$

If I suppose next iteration I will be dead, I will accept any proposal:

$$\mathbf{E}(pirate(Grade, Vote_1, Coins_1, 1), T_1) \wedge T_2 = T_1 + 1 \wedge \\ \mathbf{E}(pirate(Grade, Vote_2, Coins_2, 0), T_2) \rightarrow Vote_1 = 1.$$

The predicate $pirates(L_{coins}, L_{votes}, T)$ is the program entry point. Its arguments are the coins assignment (list L_{coins}), the result of the voting (list L_{votes}), and the iteration number T (initially 1). In the following code, CLP predicates are underlined. The *abduce* predicate abduces the atom (3) for each pirate.

```
pirates([],[],T):- npirates(N), T>N.
pirates(L_coins,L_votes,T):- npirates(N), ncoins(M), T ≤ N,
  % Define variables' domains
  length(L_coins,N), domain(L_coins,0,M), sumlist(L_coins,M),
  length(L_votes,N), domain(L_votes,0,1), sumlist(L_votes,N_votes),
  2N_votes> N-T+1 ⇔ Win, %One wins if he gets majority
  % The pirate gets the coins only if he wins
  nth(T,L_coins,CoinsPirate), GotCoins = Win*CoinsPirate,
  % The proposer will be alive only if he wins
  length(L_alive,N), nth(T,L_alive,Win),
  maximize(
  (  T1 is T+1, pirates(_,_,T1),
     abduce(L_coins,L_votes,L_alive,N,T),% Abduce a division
     labeling(L_coins), labeling(L_votes),
  ),GotCoins). %Maximise number of obtained coins
```

The result for $N = 4$ pirates and $M = 9$ coins is the following:

```
E(pirate(4,1,7,1),1) E(pirate(3,0,0,1),1) E(pirate(2,1,1,1),1) E(pirate(1,1,1,1),1)
E(pirate(4,0,0,0),2) E(pirate(3,1,9,1),2) E(pirate(2,1,0,1),2) E(pirate(1,0,0,1),2)
E(pirate(4,0,0,0),3) E(pirate(3,0,0,0),3) E(pirate(2,1,0,0),3) E(pirate(1,0,9,1),3)
E(pirate(4,0,0,0),4) E(pirate(3,0,0,0),4) E(pirate(2,0,0,0),4) E(pirate(1,1,9,1),4)
```

Pirate 4 (first row of the table) takes 7 coins for himself, gives 1 coin each to pirates 1 and 2, and nothing to pirate 3. He is sure to get 3 votes: his own, plus those of pirates 1 and 2. How can he be so sure of surviving? Because if he dies (second row), pirate 3 gets all the money, while 1 and 2 get nothing, and nevertheless pirate 2 votes for the proposal! In fact, in iteration 3, pirate 2 is sure to die: whatever proposal he makes, pirate 1 will vote against, getting in the last iteration all the money, and making pirate 2 die.

Besides the correct game theory result, this example shows remarkable features of $\mathcal{S}$CIFF. First, a $\mathcal{S}$CIFF program is a real CLP(FD) program. The user is not restricted to a subset of the available constraints, and, in particular, she can use global constraints (e.g., `sumlist`) in the knowledge base. Second, we have recursion through the optimisation meta-predicate `maximize`. $\mathcal{S}$CIFF tightly integrates CLP(FD) and abduction, thanks to its *CHR* implementation. Finally, $\mathcal{S}$CIFF is efficient: it took 49s to solve the above problem with $N = 4$ pirates on a Pentium M715, 1.5GHz, 512MB RAM computer, which is reasonable considering that the problem is at the fourth level of the polynomial hierarchy.

REFERENCES

[1] S. Abdennadher and H. Christiansen, 'An experimental CLP platform for integrity constraints and abduction', in *FQAS 2000*, pp. 141–152.

[2] M. Alberti, F. Chesani, D. Daolio, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni, 'Specification and verification of agent interaction protocols in a logic-based system', *Scalable Computing: Practice and Experience*, **8**(1), 1–13, (2007).

[3] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni, 'Verifiable agent interaction in abductive logic programming: the *S*CIFF framework', *ACM Trans. on Computational Logic*, **9**(4), (2008).

[4] H. Christiansen and V. Dahl, 'HYPROLOG: A new logic programming language with assumptions and abduction.', in *ICLP*, (2005).

[5] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni, 'The CIFF proof procedure for abductive logic programming with constraints', in *JELIA 2004*, eds., J. Alferes and J. Leite, volume 3229 of *LNAI*, (2004).

[6] K. Eshghi and R. Kowalski, 'Abduction compared with negation by failure', in *ICLP'89*, eds., G. Levi and M. Martelli, pp. 234–255, (1989).

[7] T. Frühwirth, 'Theory and practice of constraint handling rules', *Journal of Logic Programming*, **37**(1-3), 95–138, (October 1998).

[8] T. Fung and R. Kowalski, 'The IFF proof procedure for abductive logic programming', *Journal of Logic Programming*, **33**(2), (1997).

[9] M. Gavanelli, E. Lamma, P. Mello, M. Milano, and P. Torroni, 'Interpreting abduction in CLP', in *APPIA-GULP-PRODE Joint Conf. on Declarative Programming*, Reggio Calabria, Italy, (2003).

[10] A. Kakas, R. Kowalski, and F. Toni, 'The role of abduction in logic programming', in *Handbook of Logic in Artificial Intelligence and Logic Programming*, eds., D. Gabbay, C. Hogger, and J. Robinson, (1998).

[11] A. C. Kakas, A. Michael, and C. Mourlas, 'ACLP: Abductive Constraint Logic Programming', *J. of Logic Programming*, **44**(1-3), (2000).

[12] A. C. Kakas, B. van Nuffelen, and M. Denecker, '$\mathcal{A}$-System: Problem solving through abduction', in *IJCAI-01*, ed., B. Nebel, (2001).