

# Applying Coupled Resolution Engines to Knowledge Bases

H. Taylor and M.H. Williams  
Heriot-Watt University, Scotland

## Introduction

As part of Alvey project IKBS 90, a software architecture has been developed that couples a concurrent logic programming or CLP engine with a multi-threaded Prolog engine. Concurrency among mutually invoking Prolog and CLP computations is sustained partly through and-parallel execution of CLP computations on the CLP engine, and partly by executing multiple Prolog processes and a CLP engine under Unix on a multi-processor or a uni-processor. Communication among the multiple computations is realised by message passing, enabling the architecture to be multi-processed with a low degree of contention for shared memory. The coupled architecture supports multi-user knowledge based systems without compromising on the expressive power and performance of Prolog and CLP languages, and without diminishing the capacity for multi-processing at a task apposite grain of parallelism.

This approach has three advantages over an earlier proposal by Clark and Gregory [2] for a hybrid system for executing Prolog and the CLP language Parlog together. The proposal

- facilitates efficient multi- processing by not coupling the engines using shared memory
- keeps the process interpretation of CLP execution intact by not letting messages passed by variable bindings be revocable
- retains efficiency in the CLP engine by not adding the complex machinery for sustaining chronological backtracking to it

Several interfaces between Prolog and CLP computations are defined, and programs using these interfaces are given that show how this approach to coupling resolution engines can effectively combine Prolog's ability to handle deduction over knowledge bases with a CLP language's ability to program concurrent systems. The combination of these two capabilities allows simple and expressive programming of multi-user knowledge based systems in logic.

## A Coupled Architecture

CLP languages are apt for programming asynchronous and concurrent system interactions. Programs are represented by guarded definite clauses, resolution over them is done concurrently and involves committed clause choices, and variable bindings are irrevocably made. On the other

hand Prolog is apt at supporting exhaustive solution searches of stratified knowledge bases. Programs are represented by definite clauses, resolution over them is sequential and exhibits don't know non-determinism, and bindings to variables are made on a trial basis in the search for the right satisfier. The difference of approach to the revocability of bindings to variables, to commitment in clause choice, and to the capacity for parallel execution makes for quite different styles of implementation and language.

Each language excels where the other language has marked limitations. CLP languages are poor at supporting exhaustive search, and Prolog is poor at programming concurrent systems. However, both CLP-like systems programming capabilities and Prolog-like exhaustive solution searches of knowledge bases are required for fully concurrent multi-user knowledge based systems. The coupled architecture aims to reconcile their differences in a fully concurrent scheme for harnessing these resolution engines to each other. This would enable a multi-user knowledge based management system to be programmed in the CLP language, which could invoke concurrently multiple exhaustive searches over a knowledge base represented in Prolog, and thereby realise a multi-user knowledge based system in logic.

The coupled architecture consists of

- CLP language engine
- Prolog engine
- Unix framework

The CLP engine supports the Parlog control meta-call, allowing different CLP computations to execute within separate meta-calls with a fine grain of concurrency. The CLP engine may be realised by a single Unix process or by several tightly coupled Unix-like processes sharing memory.

The Prolog engine should be WAM based with extensions for escape exception and delayed goal handling. On a uni-processor it should be able to execute in a multi-threaded fashion, using lightweight SUN OS 4.0 or equivalent processes that access a common symbol table and code area for execution over a shared database, or using separate Unix processes for execution over non-shared databases.

Mutual invocations of each language through transient and boundary communicating styles of interface allow CLP and Prolog processes to invoke each other synchronously and concurrently. The whole configuration of Unix processes supporting logic programming computations can be multi-processed or executed on a single processor.

A prototype of much of the architecture has been developed in C to run on Sun workstations. The CLP engine executes full GHC and Parlog, can invoke multiple Prolog-X processes, and uses Prolog-X's interface to a Prolog Database Machine, Massey et al [3], for resolution over large scale clause databases. Alvey project IKBS 37 produced the Prolog Database Machine. It consists of a software server using various indexing strategies and a VME board extension to a Sun in order to speed retrieval by head unification, of large numbers of clauses stored on disc, Wong and Williams [8].

Multi-user knowledge based applications can be developed in the high level logic programming facilities of the coupled system, and the interface to the Prolog Database Machine enables the range over which the system can deduce to be extended to very large knowledge bases.

## **A Coupled Prolog-CLP System**

In the coupled architecture four interfaces connect Prolog and CLP computations. The environments of each computation are kept separate. All communication across interfaces is irrevocable, and can be realised by copying across the computation's current version of the goal. If the copies cannot unify, the invoked computation fails with normal goal failure consequences for the invoking environment. A high level scheme for implementing these interfaces is developed in [5]. Two forms of invocation, that cannot be resatisfied, allow Prolog to call a CLP computation, namely

call to CLP relation

- communication at boundaries
- synchronous

call to *clp(Goal)*

- communication transiently
- asynchronous

Goals in Prolog, that are not defined in the database and are not calls on primitives, invoke the CLP engine synchronously, passing values only at the start and end of invocation without any concurrency between the calling computation and the computation that is called. Input matching suspension on invocation takes place if needed. Calls in Prolog to *clp/I* invoke the CLP engine asynchronously. Argument values may be passed transiently during invocation subject to the following restriction on shared variables

## No Complex Term CLP Bindings Ban

CLP execution may not bind variables shared with Prolog to complex terms during execution.

A CLP computation is only allowed to bind shared variables to simple terms when executing on behalf of a *clp/1* call. The interface is responsible for detecting the attempt to communicate to Prolog the binding of a shared variable to a complex term. This restriction on transient communication is essential because there is nowhere appropriate in a canonical Prolog engine to store transient bindings. *clp/1* goals in Prolog execute concurrently with clause body sibling goals. The parent goal of the spawned call to the CLP engine is not be able to exit until the *clp/1* call has succeeded or failed. When a *clp/1* goal fails, Prolog backtracks to resatisfy the parent goal. The functionality provided by *clp/1* goals is like the deterministic *::/2* fork primitive proposed in Clark and Gregory [2], but does not introduce a new conjunction operator.

Each call to a CLP computation is over the same guarded clause database of a single CLP engine. It executes in its own meta-call environment, and can only be linked to the environment of another CLP meta-call via a common Prolog computation.

CLP computations have two interfaces for calling Prolog, viz

call to Prolog relation

- asynchronous
- communication at boundaries

call to *prolog(Goals)*

- asynchronous
- communication transiently

Both interfaces allow the CLP computation to execute concurrently with the Prolog engine. CLP goals that are not calls on primitives and are not defined by a guarded clause relation, invoke the Prolog engine. The values of the invocation are passed at the boundaries of invocation. No suspension on invocation takes place. Calls from within a CLP computation to *prolog/1* invoke Prolog asynchronously in a fully concurrent fashion. Input and output bindings are communicated transiently except that the *No Complex Term CLP Bindings Ban* applies to the CLP computation executing concurrently with *prolog(Goals)*. The interface is responsible for detecting the attempt to communicate to Prolog shared variables bound to complex terms by the CLP computation. The call to *prolog/1* suspends on its argument if it is unbound.

To allow successive calls to be made to the same Prolog database state and to allow databases to be shared or not shared among concurrently executing Prolog computations, the following rule is observed

### **One Database per CLP Meta-call rule**

Each CLP meta-call is associated with a distinct persistent Prolog database.

This simple rule can be implemented on a uni-processor realisation of the coupled architecture by associating one Unix process executing Prolog with each CLP meta-call. Only control meta-calls like Parlog's *call/3* that recursively invoke the CLP engine and not simple meta-calls like *call/1* count for this purpose. The process is created by the first Prolog invocation within the meta-call, and although separate execution threads (lightweight processes under Sun OS 4.0) may be invoked over the one database state within the one Unix process, only one process with one database state persists and can be accessed from the meta-call until the meta-call ends.

## **CLP Engine Applications**

The prototype of the CLP engine uses the medium of a lingua franca to support Parlog and full GHC. The lingua franca supports its own programming style as well as those of Parlog and GHC. Both full GHC and Parlog can be translated to the lingua franca and back again, Taylor [6]. The lingua franca is a modeless, purely and-parallel, committed choice language, that uses input matching on head arguments, sequences head argument matching with guard evaluation and supports Parlog-style sequential search. It embodies extensions beyond both GHC and Parlog to support various forms of unification that cannot be supported in either language, and to allow Prolog style management by side effects of its clause database. A major difference is that lingua franca primitives can be multi-moded and output to arguments of primitives is always done by atomic unification with no bindings in the event of failure. A meta-interpreter of the lingua franca invoked by the goal *meta\_clp(Goals)* is

```
meta_clp((A, B))  :- true |
                  meta_clp(A),
                  meta_clp(B).
meta_clp(A)       :- primitive(A) |
                  call(A, yes) ;
meta_clp(A)       :- true |
                  clauses(A, Cls),
                  reduce(A, Cls, B),
                  meta_clp(B).
```

```

match(G, H, Guard) :- H <= G |
                    call(Guard, yes).

reduce(G, [(H:-Gd|B)|Cls], B1) :- match(G, H, Gd) |
                                 B = B1.
reduce(G, [(_:-_)|Cls], B1) :- reduce(G, Cls, B) |
                               B = B1.
reduce(G, [((H:-Gd|B);_)|Cls], B1):- match(G, H, Gd) |
                                     B = B1 ;
reduce(G, [(_;C)|Cls], B1) :- reduce(G, [C|Cls], B) |
                              B = B1.

```

The first clause of *meta\_clp/1* handles conjunctions, which are always executed in and-parallel. The second clause uses *primitive/1* to recognise primitives and evaluates them with the meta-call *call/2*, unifying its second argument to yes or no on the meta-call's success or failure. The third clause of *meta\_clp/1* handles resolution. Clauses retrieved in the second list argument of *clauses/2* define the relation of the first argument goal, and have their selectability tested by *match/3*. The first clause of *reduce/3* examines whether the leading clause can reduce the goal. The second clause searches the next clause in parallel. The third clause examines the last clause in a group of clauses to be searched in parallel, and the fourth clause continues clause search after all previous clauses have been found unable to reduce the goal. The clause for *match/3* ensures that head matching is performed before the guard is satisfied. The primitive *<=/2* performs one-way unification, ensuring that unifying its two arguments does not bind or share right hand argument variables. The conciseness of the meta-interpreter demonstrates the lingua franca's expressive power.

A simple interface for a Prolog engine *wam/2* can be realised in the CLP engine

```

wam(yes, F) :- write('yes\n| ?-', F) |
              read(Goals, File),
              solve(Goals, R),
              wam(R, File).
wam(no, F) :- write('no\n| ?-', F) |
             read(Goals, File),
             solve(Goals, R),
             wam(R, File) .

solve(G, R) :- nonvar(G) | horn(G, R).

```

The primitive *read(G, File)* unifies *G* with the result of reading a term asynchronously from *File*. The call to *solve/2* suspends until its first argument is bound and then uses boundary

communication to invoke a Prolog relation *horn/2*

```
horn(G, yes) :- call(G).  
horn(G, no).
```

Several instances of Prolog can be run concurrently on multiple terminals by opening these terminals as files via the CLP goal *open(Terminal)*, and executing in and-parallel the goal *wam(yes, Terminal)* for each Terminal.

This simple example of a multi-user Prolog system organised within the framework of a CLP-Prolog program illustrates how the coupled architecture can be programmed to support multiple interfaces on different terminals that invoke separate Prolog execution threads. If each Prolog execution thread realises a knowledge base inference engine, and if each CLP front-end to that thread handles the interaction with the user, the whole can realise a multi-user knowledge based system.

Boundary communication is useful where premature communication using a transient interface is undesirable. For example, the CLP call

```
| ?- prolog(try(A)), check(A).
```

where *try/I* is defined in Prolog by a generate and test pair of goals

```
try(A) :- generate(A), test(A).
```

risks having a premature binding for *A* created by satisfying *generate(A)*. If this binding is communicated early to the CLP computation, and the Prolog goal *test(A)* subsequently fails for that binding, then *A* is restored and may be rebound to a new value in resatisfying *generate(A)*. Should the new value satisfy *test(A)*, the interface will detect an inconsistency when the final value for *A* is communicated, and has to fail the *prolog/I* call on a semantic exception. If instead the original call had been

```
| ?- try(A), check(A).
```

then the boundary nature of the communication means that a binding for *A* would not have been communicated until *try(A)* had succeeded.

Concurrent calls can be made over the same Prolog database state using the *One Database per Meta-call* rule, e.g.

```
| ?- horn(a(A, B)), try(A).
```

These two boundary communication calls to Prolog execute concurrently over the same database state. Calls to two different Prolog database states can be achieved by using distinct control meta-calls. For example, the call

```
call(try(A),_,_), call(try(B),_,_).
```

executes the two *try/1* calls over separate Prolog databases in separate Unix processes. When the Prolog engine is coupled to a suitable Database Machine for handling concurrent access to shared clauses, even these distinct concurrent Prolog processes can share parts of their database as demonstrated below.

## Prolog Engine Applications

The Prolog engine supports multiple concurrent Prolog computations. Each Prolog computation can be used to conduct an exhaustive search over a stratified knowledge base, e.g.

```
parent(john, mary).
parent(mary, tim).

ancestor(A, B) :- parent(A, B).
ancestor(A, C) :- parent(A, B),
                  parent(B, C).
```

by using a classic all solutions predicate like *findall/3*.

```
| ?- findall(X, ancestor(X, tim), L).
```

```
L = [mary, john]
```

Two significant extensions to Prolog are needed for realising the interfaces to CLP computations by the methods described in Taylor [5].

(a) Delayed goal handling is needed for handling communication with CLP computations. It can be realised by a minor variation on an efficient implementation of the *freeze/2* predicate, Carlsson [1].

(b) Escape exception handling is also needed to deal with system and user defined escape exceptions in Prolog, Taylor [4]. An escape exception is propagated by special backtracking that continues past ordinary choice points to the first exceptional choice point of the raised exception's name. Such a choice point is created by the primitive *otherwise/2*. Its first argument is an atomic name or a variable which unifies with the first exception raised. Its second argument is the goal alternative to the rest of the clause body that is to be executed instead. A user can raise an escape

exception using *fail(Name)*, which returns control to the last choice point for *Name*. Alternatively the system can raise a built in exception. Escape exceptions are needed for failing Prolog computations that execute concurrently with spawned *clp/1* goals that fail.

Escape exception handling is also useful for expanding the scope of Prolog's knowledge base handling capabilities beyond stratified databases to cope with recursion. It allows non-terminating recursions to be handled gracefully. The following clauses for *any/3* define a multiple solutions predicate like *findall/3* that copes with non-terminating recursions

```
any(X, G, L) :- assert(store(M-M)),
               more(X, G, L).

more(X, G, L) :- otherwise(stack_overflow, p(L)),
                 call(G),
                 collect(X),
                 fail.

more(X, G, L) :- retract(store(L-[])).

p(L) :- retract(store(L-)), !.

collect(X) :- retract(store(A-B)),
              B = [X|C],
              assert(store(A-C)), !.
```

The predicate *any/3* takes a solution element template *X*, a goal *G* and unifies its argument *L* with a list of all satisfiers of the form *X* for each satisfier of *G*. It initialises the difference list of solutions and then calls *more/3*. *more/3* generates successive solutions by backtracking using *collect/1* to add new solutions to the stored difference list.

The multiple solutions predicate *any/3* collects solutions like *findall/3* but where local stack overflow occurs, as it would on the attempt to satisfy a subgoal that searches depth-first down an infinite branch of the SLD resolution tree, Prolog raises the built in exception *stack-overflow*. This results in failure back to the last exceptional choice point for that exception and its alternative *p(L)* is executed instead. The effect of exceptional backtracking is to free much of the space claimed on the local stack and to enable execution to continue using the list of solutions computed so far. The fact that the computation has not been able to complete is indicated by the unbound end of the list of solutions in *any/3*. In this way escape exception handling allows graceful recovery from recursion loops with non-stratified databases.

Using *clp/1* Prolog can set up concurrent interfaces to CLP computations

```

par([F|Fs]) :- clp(user(F)),
              par(Fs).

par([]).

```

where *user/I* is defined in CLP as

```

user(F) :- open(F) | ask(yes, F).

ask(yes, F) :- write('yes\n| ?-', F)
              read(G, F),
              call(G, Result),
              ask(Result, F).

ask(no, F) :- write('no\n| ?-', F),
             read(G, F),
             call(G, Result),
             ask(Result, F).

```

Over this program the Prolog query

```
| ?- par(['/dev/tty1', '/dev/tty2']).
```

would set up interfaces to two CLP computations for two terminals.

## Coupled Engine Applications

A meta-interpreter for the coupled system using all four interfaces can be built out of the meta-interpreter of the lingua franca given earlier and a Prolog meta-interpreter. *meta\_clp/1* becomes

```

meta_clp((A, B)) :- true |
                  meta_clp(A),
                  meta_clp(B).

meta_clp(A)      :- primitive(A),
                  A \= prolog(_) |
                  call(A, yes).

meta_clp(prolog(A)) :- nonvar(A) |
                      prolog(meta_prolog(A));

meta_clp(A)      :- clauses(A, [C|Cs]) |
                      reduce(A, [C|Cs], B),
                      meta_clp(B);

meta_clp(A)      :- true |
                  meta_prolog(A).

```

The new third clause handles the transiently communicating interface *prolog/1*. Execution of the guard suspends until A is instantiated. The new fifth clause handles the boundary communication interface. It is only invoked if the goal is not defined in the CLP engine. A simple Prolog meta-interpreter invoked by *meta\_prolog(Goal)* supports two interfaces to the CLP engine

```

meta_prolog((A, B))  :-  meta_prolog(A)
                        meta_prolog(B).

meta_prolog(A)      :-  primitive(A) , !,
                        call(A).

meta_prolog(A)      :-  clause(A, B),
                        body(B).

meta_prolog(A)      :-  not(clause(A, _)),
                        meta_clp(A).

body((clp_call(A, X), B))  :-  !, clp_call(A, X),
                              meta_prolog(B).

body(A)              :-  meta_prolog(A).

```

The first clause of *meta\_prolog/1* handles conjunctions. The second identifies calls to primitives with *primitive/1* and executes them with *call/1*. The third clause retrieves clause unifiers using *clause/2* and executes clause bodies using *body/1*. The last clause is only invoked if A is neither a primitive nor a user defined goal. As *meta\_clp/1* is only defined in the CLP system, the body goal invokes the CLP engine atomically.

Clauses using *clp/1* are translated from the first to the second form

```

check(A, B)  :-  test(B),
                clp(try(A, C)),
                check(B, C).

check(A, B)  :-  clp_call(try(A, C), X),
                test(B),
                check(B, C),
                data(X).

```

where the call *clp\_call/2* sets the CLP computation going, and Prolog binds X when it succeeds. In the meantime *data/1* delays further execution beyond the body of the clause until X is bound.

The failure of the CLP computation causes resatisfaction of the parent goal *check/2*. The conciseness of this meta-interpreter for the coupled system helps confirm how natural, simple and expressive the four interfaces are.

## Prolog Database Machine Applications

Prolog-X is a module based Prolog, specially adapted to exercise the facilities of a Prolog Database Machine for handling very large numbers of clauses, Williams et al [7]. Using Prolog-X as the Prolog engine of a coupled engine system gives the system large scale resolution capabilities. Prolog-X clauses can either be held in internal modules in main memory or in external modules on disc under the control of the Prolog Database Machine. External modules appear almost like internal modules, except that all accesses and updates of external modules must be performed within a transaction. A Prolog-X call

```
| ?- crs_loadmodule(db, '/tmp/db').
```

opens a module called db and associates the persistent file '/tmp/db' with it. If the file already exists, then the clauses already in it become visible in the current module. The call

```
| ?- transaction(db_query(L)).
```

tries to satisfy *db\_query(L)* over the clauses in '/tmp/db'. Should concurrent access to the same file by another user cause transaction failure, the whole transaction meta-call fails without updates to clauses in the module being committed to the external file. Successful transactions atomically commit their updates to the file on exit.

## Conclusions

The examples show how the right combinations of resolution engines in an appropriate configuration and linked to the facilities of a Prolog Database Machine can be used to program large multi-user knowledge based systems in logic. The next generation of computer systems will be knowledge processing engines, and the approach outlined here, describes one way to implement them by executing logic programs in parallel.

## Acknowledgements

Support from SERC and ICL for Alvey project IKBS 90 is acknowledged.

## References

1. M. Carlsson, *Freeze, Indexing and Other Implementation Issues in the WAM*, Logic Programming, Proceedings of the Fourth International Conference, pp. 40-58, Melbourne, Australia, (May 1987).

2. K.L. Clark and S. Gregory, *Parlog and Prolog United*, Logic Programming, Proceedings of the Fourth International Conference, pp. 927-961, (May 1987).
3. P.A. Massey, D.J. Ferbrache, and M.H. Williams, *Adapting Prolog for Knowledge Based Applications*, Department of Computer Science Technical Report, Heriot-Watt University, Edinburgh, (May 1989).
4. H. Taylor, *Escape Exception Handling in Prolog*, Technical Report 88/1, Computer Science Dept, Heriot-Watt University, (December 1988).
5. H. Taylor, *Coupling Committed and Trial Binding Resolution Engines*, Computer Science Dept, Heriot-Watt University, (June 1989).
6. H. Taylor, *A Lingua Franca for Concurrent Logic Programming*, pp. 1-20, Computer Science Dept, Heriot-Watt University, (April 1989).
7. M.H. Williams, G. Chen, D.J. Ferbrache, P.A. Massey, S. Salvini, H. Taylor, and K.F. Wong, *Prolog and Deductive Databases*, Knowledge-Based Systems, pp. 188-192, (June 1988).
8. K.F. Wong and M.H. Williams, *Design Considerations for a Prolog Database Engine*, Proceedings of the Third International Conference on Data and Knowledge Bases, Jerusalem, Israel, (June 1988).