

# **Kelpie: a Concurrent Logic Programming System for Knowledge Based Applications**

Hamish Taylor

Department of Computer Science

Heriot-Watt University

## **Abstract**

A software architecture that interfaces a concurrent logic programming system to a Prolog database machine is described. The concurrent logic programming system connects a guarded clause inference engine with a definite clause inference engine to support concurrent execution of mutually invoking guarded and definite clause programs. An interface to a Prolog database machine allows the concurrent logic programming system to invoke its clause retrieval facilities via the definite clause inference engine. The result is a concurrent system able to handle both systems programming tasks on the guarded clause inference engine and exhaustive search on the definite clause inference engine that can support sizable concurrent knowledge based applications in logic.

Keywords: stream and-parallelism, don't know non-determinism, knowledge base machine, Parlog, Prolog

## **1. Introduction**

Alvey Project IKBS 90 *Parlog on Parallel Architectures* has been concerned with the development of the concurrent logic programming language Parlog on parallel architectures. An implementation of Parlog on a multi-processor machine has been developed, a development environment for Parlog programs has been produced and an architecture suitable for linking a Parlog system to a Prolog database machine has been designed and prototyped. At Imperial College the project has been concerned with

- Parlog to DACTL compiler for Alice
- Parlog development environment and other Parlog applications

At Heriot-Watt University the project concerns

- Parlog system interfaced to a Prolog Database Machine

The Parlog to DACTL compiler is a project that has implemented the flat subset of Parlog without the control meta-call on the ALICE machine [Lam & Gregory 1987]. Parlog is compiled to an intermediate target language which in turn is compiled to the ALICE machine's run-time code. Associated with this work has been work on the Parlog Programming System, a software development environment for Parlog [Clark & Foster 1987].

The *Parlog for Knowledge Bases* part of the project at Heriot-Watt university is concerned with establishing the suitability of Parlog for use with sizable knowledge based applications. It comprises three main activities.

- development of a Parlog system
- deductive engine extension to Parlog system
- Parlog system interface to a Prolog database machine

A parallel architecture for such a Parlog system might be realised in one of two ways. Either by

concurrency	multi-tasking of Parlog's resolution
multi-processing	simultaneous co-processing of Parlog's resolution

Concurrency achieved by dynamically synchronised multi-tasking with optimisations like activity biased scheduling does not speed up processing, but it enables the multiple threads of Parlog's execution state to be processed at the same time in a responsive fashion. This is important in enabling Parlog to program the management of asynchronously interacting parts of systems and in dealing with multiple interfaces to a knowledge based system without there being undue delays in processing each interface. Multi-processing would offer real execution speed ups by enabling Parlog's concurrently executable tasks to be processed in an overlapped rather than in a context switching fashion. It would best exploit processing opportunities for Parlog but is only made possible by having access to suitable hardware. Because no such multi-processor hardware was available at Heriot-Watt university to use with the project, it was decided to attempt only a concurrent implementation of Parlog on a single processor.

Two alternative ways of achieving a single processor implementation presented themselves. Either the Parlog system could be based upon Imperial College's Sequential Parlog Machine or SPM [Gregory et al 1989] or an independent Parlog system could be developed. The advantages of a development based on SPM were

- economies of time and effort in using a working system
- established user base for an add-on package
- risk of unviable outcome to an independent system avoided

The advantages of developing an independent system were

- better primitive provision and development environment possible
- total source control without compatibility worries
- no restriction on development and innovation

SPM deviated significantly from Prolog's model in its provision of a software development environment and in its provision of primitives. The result was significantly less transparency in following the detail of execution and rather less amenable provision of primitives. The Parlog Programming System has recently ameliorated some of these problems, but SPM still lags behind typical implementations of Prolog in its transparency and primitive functionality. SPM's compilation overheads are also rather high, which makes for poor turn around in iterative program

development. Furthermore, complete control over source code was needed and that couldn't be achieved with a system like SPM that was continually evolving in answer to quite different development concerns at a distant site. For these reasons and because it was thought to be interesting to attempt something rather different, an independent development of an architecture for executing Parlog was initiated. A full bloodied implementation of this independent development has not been attempted yet. Current development work has so far only produced a working interpreter prototype of the architecture.

Parlog systems employ committed choice resolution over guarded Horn clauses to perform an incomplete search for solutions. In order that a complete search over pure Horn clauses can be performed, a Parlog system needs to employ a different inference strategy than its own. For this reason the independent Parlog system being developed at Heriot-Watt university is being coupled with an extra deductive engine to perform these complete searches in a manner to be described later. It is also being interfaced to a Prolog database machine, enabling the Parlog system to use the special facilities of the Prolog database machine to handle deduction over large knowledge spaces.

## 2. The Prolog Database Machine

The context of the *Parlog for Knowledge Bases* work is defined by an associated Alvey research project IKBS 37 at Heriot-Watt university that has constructed a *high-speed multi-user Prolog Database Machine* [Massey et al 1989]. Its objective has been to construct an integrated Prolog system capable of handling very large sets of clauses. The Prolog system stores some clauses in main memory and large numbers of clauses on disc, and special hardware and software is used to accelerate retrieval by head unification of goal matching clauses stored on disc. The system is able to handle resolution over a million clauses of the same relation name and arity. Multiple logic programming systems are allowed concurrent access to these clauses. Apart from access constraints caused by concurrent use of the same set of clauses, access to all clauses is transparent to a connected Prolog system.

Several applications have been developed for the Prolog Database Machine and these form the basis of related Alvey projects.

Logic Database	IKBS 85
Expert Systems Interface	IKBS 85
Parlog Demonstrator Applications	IKBS 90

A logic database system written in Prolog uses the extended clause handling facilities of the Prolog Database Machine to manage deduction over significant amounts of knowledge [Williams et al 1988]. An expert systems interface in turn uses the facilities of the logic database to handle access to large sets of expert system rules [Salvini 1989]. Finally demonstrator programs using the Parlog interface will show how multi-user logic database management systems can be programmed in Parlog and perform concurrent retrievals through the Parlog system's deductive engine extension and its interface to the Prolog Database Machine.

The two main clause retrieval components of the knowledge base part of the Prolog Database Machine are

Clause Retrieval Server	software server for fetching clauses for client systems
Clause Retrieval Hardware	hardware filter for fetching clause unifiers from disc

By separating the functionality of clause retrieval from a logic programming system and embedding it in a quasi-autonomous server, it becomes possible to provide a standardised interface to it, that can be used to handle clause retrieval for several different clients. The two main clients are

Prolog-X+	Prolog client
Kelpie	concurrent logic programming client

Prolog-X is an abstract machine based implementation of Prolog [Clocksin 1985]. Prolog-X is compiled to an abstract machine code which is emulated in software. It was chosen for development as the Prolog Database Machine's target Prolog because of the project's collaboration with ICL. Kelpie is the name for the concurrent logic programming system being discussed here that is being developed for use with the Prolog Database Machine.

Once the retrieval functionality was separated off into an independent server, it became possible to configure the Clause Retrieval Server [Massey et al 1989] to support

- concurrent access by multiple clients
- distributed operation
- knowledge base sharing
- persistent clause storage
- transaction based access arbitration

One server could support clause retrieval requests from several different clients concurrently using request scheduling with a single thread of control. Furthermore there was no requirement for the Clause Retrieval Server to reside on the same machine as its clients. Access across a Local Area Network could be provided almost as easily. Delocalisation made it worthwhile to allow the Clause Retrieval Server to split up its retrieval responsibilities into a number of servers connected across the network, each able to handle retrieval requests by clients and by other servers. Thus retrieval requests that were not able to be handled directly on one server could be passed on to the relevant server for that relation elsewhere. Because the server stored clauses on disc and organised full access to them, the Prolog clauses held under its charge need not be reconsulted every Prolog session. Once these clauses had been put under the clause retrieval server's charge, they could be accessed only by making visible relevant relations. By giving clauses to a clause retrieval server in this way, client Prolog systems are able to function as persistent Prolog systems. Multiple clients interrogating a single server might want common access to a single set of clauses. This created no difficulty where only retrievals were concerned, but where clients wanted to update shared sets of clauses, it became important to support locking and access arbitration to maintain consistency in clients' views of shared sets of clauses. Thus the clause retrieval server was given full control over concurrency and clause update management using a locking scheme.

Each clause retrieval server is able to manage a software search for clause satisfiers of a goal over clauses stored on disc. A variety of indexing methods are employed to accelerate this. However, the main use of clause retrieval servers is to serve as software means of invoking specialised clause retrieval hardware for accelerating clause retrieval from disc. ICL's Content Addressable File Store is one possible hardware engine for performing this task. A more specialised device is the CLARE engine [Wong & Williams 1988]. The main features of the CLARE clause retrieval hardware are

- VME bus based co-processor searching on-the-fly
- two stage filter - superimposed codeword indexing & partial test unification

A backend co-processor performs its search on-the-fly as clause data streams off disc. Two filters are employed for recognising clauses whose heads unify with a call. Either filter can be used independently or both can be employed in the given order. The first filter employs superimposed codeword plus mask bits matching [Ramamohanarao & Shepherd 1985]. A superimposed codeword is associated with each clause and matched codewords cause their corresponding clause to be retrieved directly from disc. The second filter performs tests directly on clause data to see whether unification between the goal and a clause head is possible. Both filters yield false positives but neither filter yields false negatives. These false positives are eliminated by the clause retrieval server before it passes retrievals back to its client. The filters are designed for use with a VME bus. Each fills a standard VME board.

A configuration of client logic programming systems, clause retrieval servers and hardware might look like Figure 1. Multiple clients, several Prolog-X+s and a Kelpie system, are using a network of clause retrieval servers linked to several pieces of hardware. The whole forms a distributed multi-user knowledge base machine.

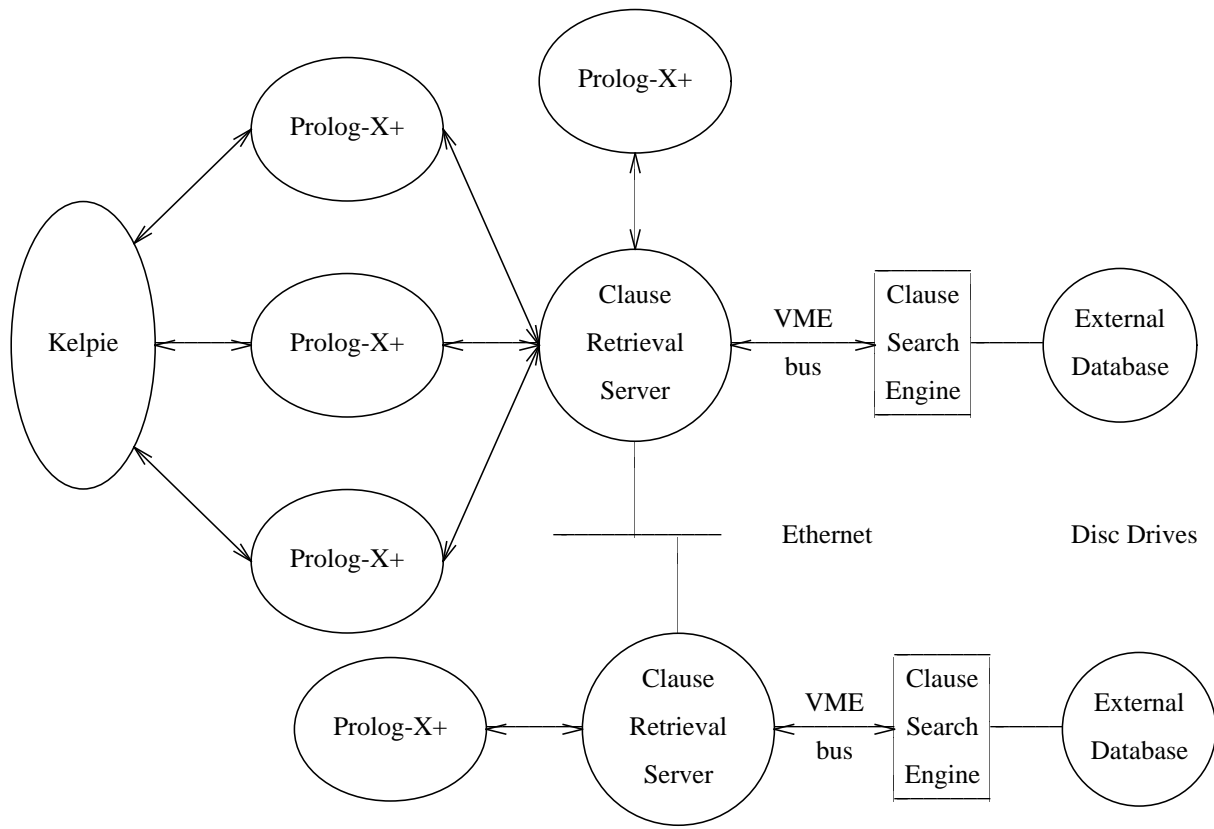


Figure 1

### 3. Forms of Resolution

The Kelpie system implements the execution of Parlog (and GHC) by means of a high level logic programming language to and from which Parlog and GHC is translated [Taylor 1989 chapter 4]. The simple nature of the translation enables the emulator of the implementation language to execute GHC and Parlog transparently. The details of the design of this emulator have been dealt with extensively elsewhere [Taylor 1989 chapter 5]. This still leaves the question as to how the deductive engine extension to the Parlog system is to be achieved. Understanding how this question is resolved requires stepping back and looking at the overall issues of concurrency and resolution strategies in relation to logic programming. Resolution affords two main kinds of opportunities for concurrent or parallel execution

- or-parallelism      computing alternative solutions in parallel
- and-parallelism    computing different parts of one solution in parallel

Except for common ancestor binding environments or-parallel computations are always independent whereas and-parallel computations can be either

- independent      no shared variables
- dependent        shared variables

stream                      dependent and-parallelism with dataflow constraints

Concurrent logic programming languages support both independent and dependent and-parallelism and allow general dataflow constraints to synchronise dynamically dependent and-parallelism. To achieve this effect efficiently, these languages limit or-parallelism to parallel evaluations of alternative guards. The result is the powerful programming technique of stream and-parallelism that is indispensable for supporting general systems programming. Canonical implementations of Prolog support neither or-parallel nor and-parallel execution. However, they are able to support co-routining, independent and-parallelism achieved by multi-tasking, and they can delay evaluation of goals in an asynchronous and data driven fashion. Recent logic programming schemes have extended the capabilities of Prolog to support general or-parallel execution [Warren 1987, Butler et al 1986, Westphal et al 1987]. Other schemes combine independent and-parallelism with backtracking using tests at run-time to determine whether conjoined literals are independent of each other or not. These tests are based upon data flow graphs constructed at compile time, that are used to determine which literals can be solved in parallel [DeGroot 1984] and which goals need to be redone when backtracking upon goal failure [Lin et al 1986, Conery 1987]. Yet further schemes [Kale 1987] aim at formulating execution models other than the And-Or process model in the hope of revealing as much scope as possible for both or-parallel and and-parallel execution of logic programs.

As well as exhibiting various opportunities for concurrent execution, resolution strategies adopted by logic programming implementations exhibit different kinds of non-determinism [Kowalski 1979]. Such non-determinism can be distinguished as being either

don't know              not known which alternative clause to pick to resolve goal with  
don't care              not cared which clause to pick to resolve goal with

Prolog mostly exhibits *don't know* non-determinism using backtracking to recover from resolutions that fail to succeed in reducing a goal. Concurrent logic programming languages like GHC and Parlog mostly exhibit *don't care* non-determinism by committing a goal to being reduced to the body of a clause once that choice is initially made. Prolog is also able to exhibit *don't care* non-determinism using the cut, which prunes considerations of further alternatives. Concurrent logic programming languages with deep guards are also able to exhibit a limited amount of *don't know* non-determinism. Deep guard searches can be used to determine which clause to select to reduce a goal, but these evaluations must be careful not to instantiate the goal without being used in its reduction.

The two kinds of non-determinism shape or-parallel opportunities for execution. Where the non-determinism is *don't know* then the alternative clauses can be resolved with in or-parallel in the search for a single goal satisfier. Or where the roots of the *don't know* non-determinism lie in the fact that more than one solution is wanted, this can also be realised in an or-parallel fashion. Where the non-determinism is *don't care*, then alternative clauses need not be considered at all and or-parallel execution of the choice is not wanted.

However, concurrent logic programming or CLP languages limit their ability to use *don't know* non-determinism by making committed bindings and committed clause choices. These committed bindings and committed clause choices

limit the ability to search for other satisfiers of a goal, and by doing so limit the possibility for a complete search. This in turn limits the usefulness of concurrent logic programming languages for programming knowledge based systems, because solutions to queries over knowledge bases must be based upon a complete search and are often required to produce every solution. Committed clause choices are likely to preclude a complete search for solutions and definitely preclude being able to produce multiple solutions.

Stream and-parallelism is necessary for programming concurrent systems and resolution exhibiting don't know non-determinism is necessary for supporting exhaustive search in knowledge based applications. Thus both are necessary for supporting concurrent knowledge based systems. The problem is how to square their conflicting requirements. Proposals for combining don't know non-determinism with stream and-parallelism are

- compile away exhaustive search in CLP language [Ueda 1986, Tamaki 1987]
- interpret exhaustive search in CLP language [Shapiro 1987, Clark & Gregory 1985]
- compile away stream and-parallelism in Prolog [Naish 1988]
- devise integrated language [Haridi & Brand 1988, Bahgat & Gregory 1989]
- couple CLP and Prolog resolution engines [Clark & Gregory 1987]

Each approach has advantages and limitations. The consideration of their relative merits is beyond the scope of this paper. For a general discussion of these approaches see [Bahgat 1988] and [Taylor 1989 chapter 2]. The last approach has been adopted by the *Parlog for Knowledge Bases* project.

#### 4. Parlog and Prolog United

One obvious way of being able to profit from efficient implementations of both don't know non-determinism and stream and-parallelism would be to couple a CLP resolution engine to a Prolog resolution engine. Clark and Gregory first advocated this approach in detail in terms of coupling a Prolog system with a Parlog system in their paper *Parlog and Prolog United* [Clark & Gregory 1987]. They explore a number of different options for supporting such coupling. In particular they mention six types of interface. Three interfaces enable a Parlog computation to call a Prolog computation

- eager all solutions predicate `set(List^, Term?, Conj?)`
- lazy multiple solutions predicate `subset(List?, Term?, Conj?)`
- single solutions predicate `prolog_call(Conj?)`

The constructors *set/3* and *subset/3* are eager and lazy multiple solutions constructors like Prolog's *findall/3*. They deliver solutions incrementally on the List argument. The third interface *prolog\_call/1* appears as a Parlog goal to the Parlog system which can execute concurrently with other Parlog goals. It can have its variables bound either by other Parlog goals executing concurrently with it or by the Prolog computation it represents at any time during its execution. These new bindings are visible during the Prolog computation to both it and its calling environment. They may not be rescinded by the Prolog computation once made. Since the multiple solutions constructors can be supported fairly

easily using *prolog\_call/1*, the real functionality in question is represented by *prolog\_call/1* alone. Three interfaces are proposed for enabling a Prolog computation to call a Parlog computation

- deterministic conjunction            `prolog-conj :: parlog-conj`
- eager non-deterministic conjunction    `prolog-conj <> parlog-conj`
- lazy non-deterministic conjunction    `prolog-conj << parlog-conj`

Each of the three interfaces represents co-routining conjunctions. The first spawns the Parlog conjunction immediately on execution and then continues executing the Prolog conjunction. The Prolog conjunction may engage in backtracking so long as no bindings passed to the Parlog conjunction are undone. The whole meta-conjunction succeeds when both conjuncts succeed. When the Prolog conjunct is *true*, the Prolog system is just synchronously invoking a Parlog computation. The second and third interfaces allow failures in the Prolog conjunction to fail and undo bindings to variables shared with the Parlog conjunction. The result is to cause rollback in the Parlog computation to the point at which the uncommitted binding that was undone was made. If the Parlog computation itself fails, failure of the goal that caused the most recent uncommitted binding in the Prolog conjunction is supposed to be initiated. The difference between the second and the third interface is that the second interface `<>/2` allows the Prolog conjunction to carry on making uncommitted bindings to variables shared with the Parlog computation eagerly, while the third interface `<</2` makes deterministic bindings to shared variables eagerly yet suspends upon making an uncommitted binding (i.e. one it could undo on backtracking) and only proceeds if and when the Parlog conjunction deadlocks.

The motivation behind Clark and Gregory's proposals is to explore the range of possible ways of coupling Parlog and Prolog systems together with a view to stimulating further research into the design, implementation and use of hybrid don't know and don't care non-deterministic logic programming systems. The two non-deterministic interfaces `<>/2` and `<</2` represent the really radical departure in Clark and Gregory's paper, because they entail extending Parlog to allow bindings in a Parlog computation to be undone and the Parlog computation rolled back, and Clark and Gregory devote much of their attention in their paper to the non-deterministic operators. However, the motivation for the research being pursued here is not to explore the full range of possible interfaces between Prolog and CLP computations but to support a general type of application requiring concurrency - knowledge based systems. This kind of application is mostly concerned with interfaces from Parlog-like computations to Prolog-like computations. However, interfaces that enable Prolog computations to invoke CLP computations are also useful to allow knowledge based applications in Prolog to deal with concurrent interfaces.

Clark and Gregory argue for a shared memory approach to storing variables shared between Prolog and Parlog computations. Coupled Parlog and Prolog computations are allowed to bind variables in each others heaps. A different approach would be to make each Prolog and CLP computation maintain a completely separate binding environment and to copy bindings across the interfaces both ways. This would make variable management simpler, make garbage collection easier, make parallel multi-processing more efficient, and would allow more modular development of coupled systems with a lower degree of intervention into the implementation of the component resolution engines. Thus a non-shared memory approach has been adopted by the *Parlog for Knowledge Bases* project.

## 5. Coupled Resolution Engines

CLP and Prolog resolution engines couplings should be flexible and efficient, while preserving as much concurrency as possible to enable several CLP and Prolog evaluations to invoke each other and execute at the same time. However, CLP computations cannot support backtracking and Prolog computations cannot support stream and-parallelism without making major extensions to their canonical execution models that would seriously undermine their efficiency. As efficiency is crucial and as our approach takes as its starting point the idea of forming a hybrid of Prolog and CLP styles of resolution engine, neither of these extensions will be countenanced. Bindings made to variables in a Prolog computation that are shared with a CLP computation will always be committed once they are communicated.

Furthermore Prolog relation literals in a Prolog computation will always be resolved away sequentially unless they are computed asynchronously in a freeze/2 like fashion. In this way conceptual simplicity and the merits of the existing forms of implementation of each kind of resolution engine can be preserved. Clark and Gregory's two non-deterministic interfaces from Prolog to Parlog  $\langle \rangle / 2$  and  $\langle \langle \rangle / 2$  envisage backtracking in a CLP implementation. The state information that must be preserved in a CLP computation to enable this will often be large and not very tractable to manage for each *backtrack point* that must be preserved in the Parlog computation, the approach will require extensive re-design of the CLP engine to support backtracking, and any advantage of a hybrid approach like this over an integrated language approach like Andorra Prolog [Haridi & Brand 1988] or Parallel NU-Prolog [Naish 1988] would be likely to vanish. Accordingly interfaces like these will not be used.

The various needs for coupled resolution require a variety of coupling interfaces. These interfaces will operate under the basic assumption that the environments of the two resolution engines are completely separate, except when invoked, as explained below, so as to share code areas and symbol tables. All communication across interfaces will be by full unification with variable restoration on failure and involve copying terms across where a variable has been bound. If the unification fails, the invoked computation will fail with normal goal failure consequences for the invoking environment. The following two interfaces to a CLP computation from a Prolog computation are to be supported. Calls using these interfaces are never resatisfiable.

call to CLP relation

- communication at boundaries
- input argument suspension on invocation possible
- synchronous

call to *clp(Goal)*

- communication transiently both ways (with a restriction)
- input argument suspension on invocation of Goals possible
- asynchronous

Goals in a Prolog computation, that are not defined by Prolog clauses, invoke the CLP resolution engine. The invocation is synchronous, and values are passed at the boundaries of invocation without any concurrency between the

calling computation and the computation that is called. Input matching suspension on invocation may take place. This interface is similar to that proposed by Clark and Gregory's deterministic operator

```
true :: relation(Arg1, ... ,ArgN)
```

where the first conjunction is *true* and *relation/N* is a CLP relation.

In the second type of interface the *clp/I* goal in the Prolog computation coroutines with the goals after it in the body of its clause. Argument values are passed transiently during invocation subject to the following restriction on shared variables

**No Complex Term CLP Bindings**      CLP execution may not bind variables shared with a Prolog computation 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/I* call. The interface is responsible for detecting the attempt to communicate to a Prolog computation the binding of a shared variable to a complex term. This restriction has to be made because such bindings cannot be safely made at the top of the heap at the time of communication. In the event of a *clp/I* call failing, the Prolog computation backtracks and attempt to resatisfy the parent of the *clp/I* goal. The parent goal of the spawned call to the committed choice resolution engine is not able to exit until the *clp/I* call has succeeded or failed. This type of invocation performs a role similar to Clark and Gregory's deterministic `:: fork` primitive but in a more natural fashion. The following two interfaces to a Prolog computation from a CLP computation are to be supported.

call to Prolog relation

- asynchronous
- communication at boundaries
- executes concurrently with CLP resolution engine
- no suspension on invocation

call to *prolog(Goals)*

- asynchronous
- communication transiently both ways (with a restriction)
- executes concurrently with CLP resolution engine
- input argument suspension if *Goals* unbound

Goals in a CLP computation, that are not defined by a guarded clause relation, invoke the Prolog resolution engine. The invocation is synchronous, and values are passed at the boundaries of invocation without any concurrency between the calling computation and the computation that is called. No suspension on invocation takes place. Calls in a CLP computation to *prolog/I* invoke the Prolog resolution engine asynchronously in a fully concurrent fashion. Input and output bindings are communicated transiently. *prolog/I* is rather similar to Clark and Gregory's proposed interface *prolog\_call/I*, although the implementation approach is quite different. Because *prolog/I* subsumes the

functionality of *set/3* as Clark and Gregory have shown, and as reasonably functional restricted versions of *subset/3* can be realised by *prolog/1*, multiple solutions interfaces like *set/3* and *subset/3* do not need to be directly supported.

To allow flexible private access to and sharing of Prolog databases by the CLP resolution engine the following principle is adopted

**One Database per Meta-call**      each separate CLP meta-call environment is associated with a separate persistent Prolog database state.

This database state is created by the first call to a Prolog computation from within that meta-call's environment. Subsequent to that, each call to a Prolog computation within that meta-call environment accesses the same database state and once created that Prolog database state persists for as long as the CLP meta-call environment persists. Invocations of Prolog computations within different CLP meta-calls are always to different Prolog database states.

## 6. "Implementation Issues"

A multi-user knowledge based system can be supported on this sort of coupled CLP-Prolog system using one Unix process (or several tightly interacting Unix-like processes sharing memory on a multi-processor) to execute the CLP language and one or a few Unix processes per user to execute sequential Prologs. The Unix operating system sustains overall concurrency among the multiple sequential Prologs and the sole CLP engine. The CLP resolution engine is used to process the closely interacting and communicating tasks of the knowledge based system as CLP computations with fine granularity concurrency, leaving the remaining knowledge processing tasks requiring don't know non-determinism to be processed with low degrees of interaction and large granularity concurrency by one or a few separate sequential Prologs per user. The whole system is programmed by a CLP program executing on the CLP engine and separate Prolog programs executing in each Prolog process. Multiple Prolog processes executing at the same time on a single processor share code segments, and thus only one copy of the Prolog engine code is kept resident in that processor's fast memory at a time.

In order to sustain concurrent execution of multiple Prologs on a more closely coupled basis as required by the *One database per Meta-call* rule, the Prologs are also able to share common memory areas. This is realised under Sun Unix OS 4.0 using lightweight processes sharing the memory of a single Unix process. A conventional Prolog engine adapted to run as a multi-threaded Prolog engine divides up shared memory up so that these separate Prolog computation threads have their own non-virtual local, global and trail stacks and share only the code area and symbol tables using monitors to handle the critical code of updates to these areas. The lightweight process scheduler switches the engine between threads on a resource sliced basis.

The CLP resolution engine uses an and-parallel guarded clause adaptation of the And-Or tree execution model [Takeuchi & Furukawa 1986]. Three kinds of process are used. Primitive processes represent system predicates, literal processes represent goals before they are reduced, and clause processes represent clauses being examined to assess the reducibility of a goal. Processes are linked to their parent, to their siblings and to their children, and they are

either in the state of being executed, being executable or are suspended. Suspended processes are either suspended upon their children, suspended upon one or more variables, suspended upon an input stream or are suspended upon a Prolog computation. Processing is done on a *bounded depth first* basis and concurrency is achieved by resource quotas of execution cycles allowed with children inheriting their parent's quota less one. An activity bias operates in favour of recently awoken processes of being executed early [Taylor 1989 chapter 5].

The Prolog resolution engine is based upon Prolog-X, a conventional Prolog engine based upon Clocksin's Zip abstract machine for Prolog [Clocksin 1985]. Execution can be suspended on the primitive *data/1*, synchronously upon an input stream or upon a CLP computation. Processing is done upon a purely depth first basis and concurrency between lightweight Prolog processes is achieved by attaching resource quotas to reductions performed, and context switching after quotas are consumed. Prolog-X also supports two special features needed for implementing the interfaces defined above.

- delayed goal execution
- escape exception handling

In the updated version of Prolog-X used on the Prolog Database Machine [Massey et al 1989] goals may be delayed using *freeze/2* [Carlsson 1987]. At whatever point their variable is bound, they become re-attached to the computation. If they fail, they initiate backtracking at that point only. Prolog-X also features escape exception handling somewhat like that featured in Ada and ML. Named exceptional choice points can be defined that are invisible to normal backtracking. When an exception of that name is raised by the system or the user, the system backtracks directly to the most recent exceptional choice point of that exception name [Taylor 1989 chapter 6]. The implementation of the interfaces between Prolog and CLP Computations using these extensions will be the subject of a future paper.

## 7. Conclusion

A concurrent logic programming system consisting of two connected resolution engines has been described. It is being prototyped in C to run on a single processor workstation under Unix. The rationale for and a defence of the design of the system have been given and its general characteristics have been specified. The overall system looks as follows.

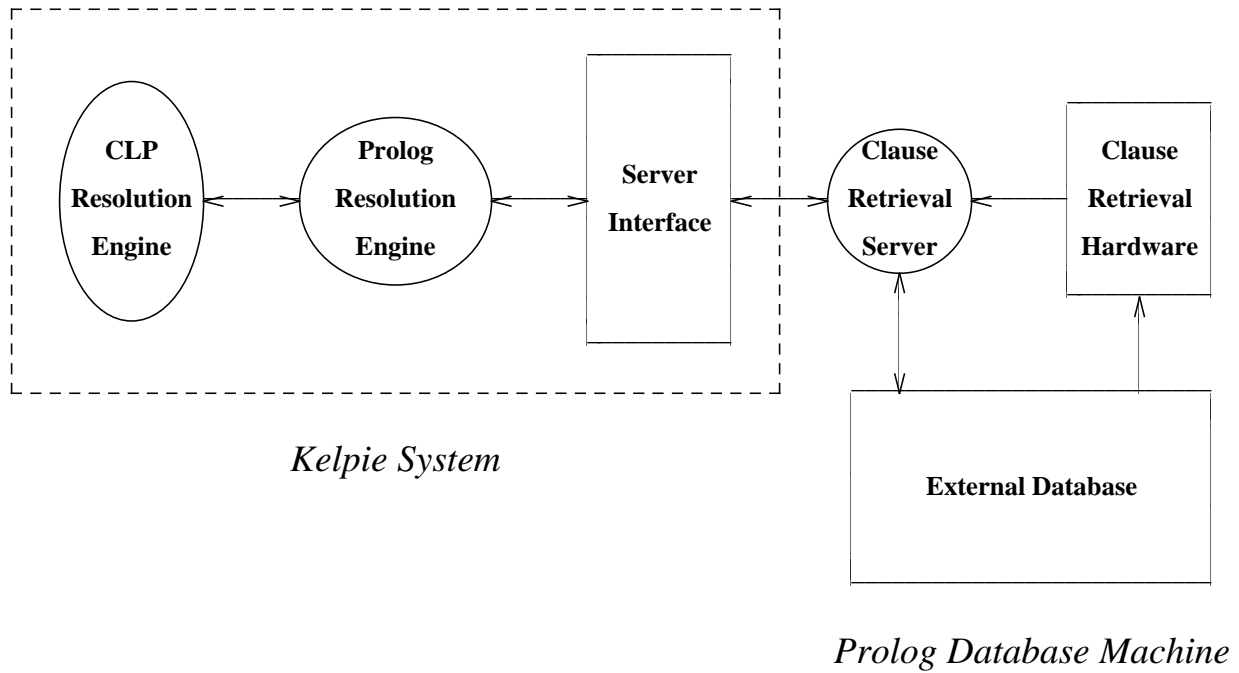


Figure 2

The CLP resolution engine is connected with a Prolog resolution engine and interfaced through it to a Prolog Database Machine. One way of demonstrating the capabilities of both parts of the connected system and its Prolog database machine interface would be to have a logic database management system with a number of interfaces on separate terminals running on the CLP resolution engine. Retrievals over the knowledge base would invoke the Prolog resolution engine concurrently which in turn would be able to exercise the facilities of the Prolog database machine. In this way the combined system would realise a concurrent logic programming system for sizable knowledge based applications by combining the systems programming virtues of the CLP engine with the complete knowledge base search capabilities of the Prolog engine.

### Acknowledgements

The author wishes to acknowledge the contributions of David Ferbrache, Paul Massey, Howard Williams and Kam Fai Wong, whose worked on the Prolog Database Machine project, the financial support of the SERC under the Alvey initiative, help from members of Imperial College's Parlog group, the support of his wife and colleagues at Heriot-Watt University and his project's industrial collaborator ICL.

## References

- Bahgat 1988 R. Bahgat, "Towards an Integrated Don't Know Stream And-parallel Logic Language", Res. Rep. DOC 88/1, Dept. of Computing, Imperial College, London, February 1988
- Bahgat & Gregory 1989 R. Bahgat, S. Gregory, "Pandora: Non-Deterministic Parallel Logic Programming", Proceedings of the 6th Int. Logic Programming Conf., Lisbon, May 1989
- Butler et al 1986 R. Butler, E. Lusk, R. Olson, W. McCune, R.A. Overbeek, "ANLWAM: a parallel implementation of the Warren Abstract Machine", Mathematics and Computer Science Division, Argonne National Lab., August 1986
- Carlsson 1987 M. Carlsson, "Freeze, Indexing and Other Implementation Issues in the WAM", Proceedings of the 4th Int. Conf. on Logic Programming, Melbourne, pp. 40-58, May 1987
- Clark & Foster 1987 K.L. Clark, I.T. Foster, "A Declarative Environment for Concurrent Logic Programming", Proceedings of TAPSOFT, Pisa, 1987
- Clark & Gregory 1985 K.L. Clark, S. Gregory, "Notes on the Implementation of Parlog", Journal of Logic Programming 2(1), July 1985
- Clark & Gregory 1987 K.L. Clark, S. Gregory, "Parlog and Prolog United", Proceedings of the 4th Int. Conf. on Logic Programming, Melbourne, pp. 927-961, May 1987
- Clocksinn 1985 W.F. Clocksin, "The Design and Simulation of a Sequential Prolog Machine", New Generation Computing, OHMSHA Ltd, pp. 101-120, 1985
- Conery 1987 J.S. Conery, "Implementing Backward Execution in Non-Deterministic AND-Parallel Systems, Proceedings of the 4th Int. Conf. on Logic Programming, Melbourne, pp. 633-653, May 1987
- DeGroot 1984 D. DeGroot, "Restricted AND-Parallelism", Proceedings of the Int. Conf. on 5th Generation Computer Systems, Tokyo, pp. 471-478, 1984
- Gregory et al 1989 S. Gregory, I.T. Foster, A. Burt, G.A. Ringwood, "An Abstract Machine for the Implementation of Parlog on Uniprocessors", New Generation Computing, 6(4), pp. 389-420, 1989
- Haridi & Brand 1988 S. Haridi, P. Brand, "Andorra Prolog: an integration of Prolog and committed choice languages", Proceedings of the Int. 5th Generation Computer Systems Conf., Tokyo, December 1988
- Kale 1987 L.V. Kale, "The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs", Proceedings of the 4th Int. Conf. on Logic Programming, Melbourne, pp. 616-632, May 1987
- Kowalski 1979 R. Kowalski, "Logic for Problem Solving", Elsevier/North-Holland, New York, 1979
- Lam & Gregory 1987 M. Lam, S. Gregory, "Parlog and ALICE: A Marriage of Convenience", Proceedings of the 4th Int. Logic Programming Conf., Melbourne, May 1987
- Lin et al 1986 Y. Lin, W. Kumar, C. Leung, "An intelligent backtracking algorithm for parallel execution of logic programs", Proceedings of the 3rd Int. Conf. on Logic Programming, London, pp. 55-68, July 1986
- Massey et al 1989 P. Massey, D. Ferbrache, M. H. Williams, "Adapting Prolog for Knowledge Based Applications", Tech. Rep, Dept. of Computer Science, Heriot-Watt University, Edinburgh, May 1989
- Naish 1988 L. Naish, "Parallelizing NU-Prolog", Proceedings of the 5th Int. Conf./Symp. on Logic Programming, Seattle, pp. 1546-1564, August 1988
- Ramamohanarao & Shepherd 1985 K. Ramamohanarao, J. Shepherd, "A superimposed Codeword Indexing Scheme for Very Large Prolog Databases", Tech Rep. 85/17, Dept. of Computer Science, University of Melbourne, 1985

- Shapiro 1987 E. Shapiro, "An Or-Parallel Execution Algorithm for Prolog and its FCP Implementation", Proceedings of the 4th Int. Conf. on Logic Programming, Melbourne, pp. 311-337, May 1987
- Salvini 1989 S. Salvini, "An Expert Systems Interface to a Logic Database", Tech. Rep, Dept. of Computer Science, Heriot-Watt University, Edinburgh, April 1989
- Takeuchi & Furukawa 1986 A. Takeuchi, K. Furukawa, "Parallel Logic Programming Languages", Proceedings of the 3rd Int. Conf. on Logic Programming, London, pp. 242-254, July 1986
- Tamaki 1987 H. Tamaki, "Stream-based Compilation of Ground I/O Prolog into Committed Choice Choice Languages", Proceedings of the 4th Int. Conf. on Logic Programming, Melbourne, pp. 376-393, May 1987
- Taylor 1989 H. Taylor, "Coupled Resolution Engines for Programming Knowledge Based Systems in Logic", PhD thesis, Heriot-Watt university, December 1989
- Ueda 1986 K. Ueda, "Guarded Horn Clauses", D. Eng. thesis, Tokyo University, March 1986
- Warren 1987 D.H.D. Warren, "The SRI model for Or-parallel execution of Prolog - abstract design and implementation issues", Proceedings of the Symp. on Logic Programming, San Francisco, pp. 92-102, August 1987
- Westphal et al 1987 H. Westphal, P. Robert, J. Chassin, J-C. Syre, "The PEPSys model: combining backtracking, and- and or-parallelism", Proceedings of the Symp. on Logic Programming, San Francisco, pp. 436-448, August 1987
- Williams et al 1988 M.H. Williams, G. Chen, D.J. Ferbrache, P.A. Massey, S. Salvini, H. Taylor, K.F. Wong, "Prolog and Deductive Databases", Knowledge Based Systems, pp. 188-192, June 1988
- Wong & Williams 1988 K.F. Wong, M.H. Williams, "Design Considerations for a Prolog Database Engine", Proceedings of the 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, June 1988