Coordination and Concurrency in Multi-Engine Prolog

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

Abstract. We discuss the impact of the separation of logic engines (independent logic processing units) and multi-threading on the design of coordination mechanisms for a Prolog based agent infrastructure.
We advocate a combination of coroutining constructs with focus on expressiveness and a simplified, multi-threading API that ensures optimal use available parallelism.
In this context, native multi-threading is made available to the application programmer as a set of high-level primitives with a declarative flavor while cooperative constructs provide efficient and predictable coordination mechanisms. As illustrations of our techniques, a parallel `fold` operation as well as cooperative implementations of Linda blackboards and publish/subscribe are described.
Keywords: *multi-engine Prolog, agent coordination, high-level multi-threading, coroutining Linda blackboards, publish/subscribe, Java-based Prolog system*

1 Introduction

Multi-threading has been adopted in today's Prolog implementations as it became widely available in implementation languages like C or Java.

An advantage of multi-threading over more declarative concurrency models like various AND-parallel and OR-parallel execution schemes, is that it maps to the underlying hardware directly: on typical multi-core machines threads and processes are mapped to distinct CPUs[1]. Another advantage is that a procedural multi-threading API can tightly control thread creation and thread reuse.

On the other hand, the explicit use of a procedural multi-threading API breaks the declarative simplicity of the execution model of logic based languages. At the same time it opens a Pandora's box of timing and execution order dependencies, resulting in performance overheads for various runtime structures that need to be synchronized. It also elevates risks of software failure due to programmer errors given the mismatch between assumptions about behavior expected

[1] We use the word CPU in accordance of what the underlying runtime system and operating system sees as independent processing units in a multi-core/multi-processor machine. For instance, on a two Xeon processor Quad-Core MacPro with hyper-threading, the Java VM sees 16 independent processing units.

to follow the declarative semantics of the core language and the requirements of a procedural multi-threading API.

In this paper we will describe how efficient and flexible agent coordination is facilitated by a design emphasizing *the decoupling of the multi-threading API and the logic engine operations and encapsulation of multi-threading in a set of high-level primitives with a declarative flavor.*

In this process, we use threads encapsulated as high level programming language constructs with focus on performance benefits, and we are resorting to determinacy, through lightweight, cooperative sequential constructs, to express coordination patterns.

We have implemented the API in the context of an experimental, Java-based Prolog system, Lean-Prolog[2].

LeanProlog is based on a reimplementation of BinProlog's virtual machine, the BinWAM. It succeeds our *Jinni Prolog* implementation that has been used in various applications [1–4] as an *intelligent agent infrastructure*, by taking advantage of Prolog's knowledge processing capabilities in combination with a simple and easily extensible runtime kernel supporting a flexible reflexion mechanism. Naturally, this has suggested to investigate whether some basic agent-oriented language design ideas can be used for a refactoring of Prolog's interoperation with the external world, including interaction with other instances of the Prolog processor itself.

Agent programming constructs have influenced design patterns at "macro level", ranging from interactive Web services to mixed initiative computer human interaction. From the very beginning, *Performatives* in Agent communication languages [5, 6] have made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At the same time, it has been a long tradition of logic programming languages [7] to use multiple logic engines for supporting concurrent execution.

In this context we have centered our implementation around logic engine constructs providing an API that supports reentrant instances of the language processor. This has naturally led to a view of logic engines as instances of a generalized family of iterators called *Fluents* [8], that have allowed the separation of the first-class language interpreters from the multi-threading mechanism, while providing a very concise source-level reconstruction of Prolog's built-ins. Later we have extended the original *Fluents* with a few new operations [9] supporting bi-directional, mixed-initiative exchanges between engines, bringing them closer to an agent-oriented view as autonomous logic processors.

The resulting language constructs, that we have called *Interactors*, express control, metaprogramming and interoperation with stateful objects and external services.

[2] It is called Lean-Prolog as we have consistently tried to keep implementation complexity under control and follow minimalist choices in the design of built-ins, external language interfaces and a layered, modular extension mechanism.

On the other hand, our multi-threading layer has been designed to be independent of the interactor API. This allows assumptions of determinacy when working with multiple engines (and other sequential interactors) within a thread.

The multi-threading API integrates thread-construction with interactors called `Hubs` that provide synchronization between multiple consumers and producers. It supports high-level performance-centered concurrency patterns while removing the possibility of programming errors involving explicit synchronization.

The guiding architectural principle we based our design on, can be stated concisely as follows: *separate concurrency for performance from concurrency for expressiveness*. Arguably, it is a good fit with the general idea behind declarative programming languages – delegate as much low level detail to underlying implementation as possible rather than burdening the programmer with complex control constructs.

The paper is organized as follows.

Section 2 overviews logic engines and describes their basic operations and the interactor API that extends the same view to various other built-in predicates. Section 3 introduces `Hubs` - flexible synchronization devices that allow interoperation and coordination between threads. Section 4 describes a set of high-level multi-threading operations that ensure concurrent execution seen as a means to accelerate computations while keeping the semantics as close as possible to a declarative interpretation.

Sections 5 and 6 show that fundamental coordination patterns like Linda blackboards and publish/subscribe can be implemented cooperatively in terms of sequential operations on logic engines.

Finally, section 7 discusses related work and section 8 concludes the paper.

2 Logic engines as answer generators

Our *Interactor API* has evolved progressively into a practical Prolog implementation framework starting with [8] and continued with [10] and [9]. We summarize it here and refer to [9] for the details of a semantic description in terms of Horn Clause Logic of various engine operations.

A *logic engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another *engine* very much the same way a programmer controls Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it.

A logic engine can be seen as an instance of the *Prolog runtime system* implementing LD-resolution [11] on a given clause database, together with a set of built-in operations. The command

```
new_engine(AnswerPattern, Goal, Interactor)
```

creates a new logic engine, uniquely identified by `Interactor`, which shares code with the currently running program and is initialized with `Goal` as its starting point. `AnswerPattern` is a term, usually a list of variables occurring in `Goal`, of which answers returned by the engine will be instances. Note however that

`new_engine/3` acts like a typical constructor, no computations are performed at this point, except for initial allocation of data areas.

2.1 Iterating over computed answers

Note that our logic engines are seen, in an object oriented-style, as implementing the *interface* `Interactor`. This supports a *uniform* interaction mechanism with a variety of objects ranging from logic engines to file/socket streams and iterators over external data structures.

The `ask_interactor/2` operation is used to retrieve successive answers generated by an `Interactor`, on demand. It is also responsible for actually triggering computations in the engine. The query

```
ask_interactor(Interactor, AnswerInstance)
```

tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`. If an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned. Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `ask_interactor/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller interpreter does not interfere with `Interactor`'s iteration over answers. Backtracking over `Interactor`'s creation point, as such, makes it unreachable and therefore subject to garbage collection. An interactor is stopped with the

```
stop_interactor(Interactor)
```

operation, that, in the case of logic engines, allows reclaiming resources held by the engine.

So far, these operations provide a minimal API, powerful enough to switch tasks cooperatively between an engine and its client and emulate key Prolog built-ins like `if-then-else` and `findall` [8], as well as typical higher order operations like *fold* and *best_of* [9].

2.2 A yield/return operation

The following operations provide a "mixed-initiative" interaction mechanism, allowing more general data exchanges between an engine and its client.

First, like the `yield return` construct of `C#` and the `yield operation` of Ruby and Python, our `return/1` operation

```
return(Term)
```

will save the state of the engine and transfer *control* and a *result* `Term` to its client. The client will receive a copy of `Term` simply by using its `ask_interactor/2` operation.

Note that an interactor returns control to its client either by calling `return/1` or when a computed answer becomes available. By using a sequence of `return`

and `ask_interactor` operations, an engine can provide a stream of *intermediate/final results* to its client, without having to backtrack. This mechanism is powerful enough to implement a complete exception handling mechanism simply by defining

```
throw(E) :- return(exception(E)).
```

When combined with a `catch(Goal, Exception, OnException)`, on the client side, the client can decide, upon reading the exception with `ask_interactor/2`, if it wants to handle it or to throw it to the next level.

2.3 Coroutining logic engines

Coroutining has been in use in Prolog systems mostly to implement constraint programming extensions. The typical mechanism involves *attributed variables* holding suspended goals that may be triggered by changes in the instantiation state of the variables. We discuss here a different form of coroutining, induced by the ability to switch back and forth between engines.

The operations described so far allow an engine to return answers from any point in its computation sequence. The next step is to enable an engine's *client*[3] to *inject* new goals (executable data) to an arbitrary inner context of an engine. Two new primitives are needed:

```
to_engine(Engine, Data)
```

that is called by the client [4] to send data to an Engine, and

```
from_engine(Data)
```

that is called by the engine to receive a client's Data.

A typical use case for the *Interactor API* looks as follows:

1. the client creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*, parameterized as follows:
   (a) the *client* passes some data and a new goal to the *engine* and issues an `ask_interactor/2` operation that passes control to it
   (b) the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
   (c) the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

[3] Another engine, that uses an engine's services.

[4] Equivalently the `tell_interactor/2` generic interactor predicate can be also used here.

3 Hubs and threads

As a key difference with typical multi-threaded Prolog implementations like Ciao-Prolog [12] and SWI-Prolog [13], our Interactor API is designed up front with a clear separation between *engines* and *threads* as we prefer to see them as orthogonal language constructs.

To ensure that communication between logic engines running concurrently is safe and synchronized, we hide the engine handle and provide a producer/consumer data exchanger object, called a `Hub`, when multi-threading.

A `Hub` can be seen as an interactor used to synchronize threads. On the Prolog side it is introduced with a constructor `hub/1` and works with the standard interactor API:

```
ask_interactor(Hub, Term)
tell_interactor(Hub, Term)
stop_interactor(Hub)
```

On the Java side, each instance of the `Hub` class provides a synchronizer between M producers and N consumers. A `Hub` supports data exchanges through a private object `port` and it implements the `Interactor` interface. Consumers issue `ask_interactor/2` operations that correspond to `tell_interactor/2` operations issued by producers.

A group of related threads are created around a `Hub` that provides both basic synchronization and data exchange services. The built-in

```
new_logic_thread(Hub, X, G, Clone, Source)
```

creates a new thread by either "cloning" the current Prolog code and symbol spaces or by loading new Prolog code in a separate name space from a `Source` (typically a precompiled file or a stream). Usually a default constructor

```
new_logic_thread(Hub, X, G)
```

is used. It shares the code but it duplicates the symbol table to allow independent symbol creation and symbol garbage collection to occur safely in multiple threads without the need to synchronize or suspend thread execution.

4 High-level concurrency with higher-order constructs

Encapsulating concurrent execution patterns in high-level abstractions, when performance gains are the main reason for using multiple threads, avoids forcing a programmer to suddenly deal with complex procedural issues when working with (mostly) declarative constructs in a language like Prolog. It is also our experience that in an exclusively dynamically-typed language like Prolog this reduces software risks significantly.

One of the deficiencies of sequential or multi-threaded `findall`-like operations is that they might build large lists of answers unnecessarily. With inspiration drawn from combinators in functional languages, one can implement a more flexible multi-threaded `fold` operation instead.

The predicate `multi_fold(F, XGs, Xs)` runs a list of goals `XGs` of the form `Xs :- G` and combines, with `F`, their answers, to accumulate them into a single final result, without building intermediate lists.

```
multi_fold(F, XGs, Final) :- hub(Hub),
  length(XGs,ThreadCount),
  launch_logic_threads(XGs, Hub),
  ask_interactor(Hub, Answer),
  (Answer = the(Init) →
    fold_thread_results(ThreadCount, Hub, F, Init, Final)
  ; true
  ),
  stop_interactor(Hub),Answer=the(_).
```

The predicate `multi_fold` relies on the predicate `launch_logic_threads` to run threads initiated by the goal list `XGs`. When launching the threads, we ensure that they share the same `Hub` for communication and synchronization.

```
launch_logic_threads([], _Hub).
launch_logic_threads([(X :- G)|Gs], Hub) :-
  new_logic_thread(Hub, X, G),
  launch_logic_threads(Gs, Hub).
```

Once all threads are launched, we use the predicate `fold_thread_results` to collect results computed by various threads from `Hub`, and to combine them into a single result, while keeping track of the number of threads that have finished their work.

```
fold_thread_results(0, _Hub, _F, Best, Best).
fold_thread_results(ThreadCount, Hub, F, SoFar, Best) :-
  ThreadCount > 0,
  ask_interactor(Hub, Answer),
  count_thread_answer(Answer, ThreadCount, ThreadsLeft, F, SoFar, Better),
  fold_thread_results(ThreadsLeft, Hub, F, Better, Best).

count_thread_answer(no, ThreadCount, ThreadsLeft, _F, SoFar, SoFar) :-
  ThreadsLeft is ThreadCount-1.
count_thread_answer(the(X), ThreadCount, ThreadCount, F, SoFar, Better) :-
  call(F, X, SoFar, Better).
```

A typical application is the predicate `multi_best(F, XGs, M)`, which runs a list of goals `XGs` of the form `N :- G` where `N` is instantiated to a numeric value. By using `max/3` to combine the current best answers with a candidate one it extracts at the the maximum `M` of all answers computed (in an arbitrary order) by all threads.

```
multi_best(XGs,R) :- multi_fold(max,XGs,R).
```

Note that, as in the case of its `fold` cousins in functional languages, `multi_fold` can be used to emulate various other higher order predicates. For instance a

`findall`-like predicate is emulated as the predicate `multi_all(XGs,Xs)` which runs a list of goals `XGs` of the form `Xs :- G` and combines all answers to a list using `list_cons`.

```
multi_all(XGs, Rs) :- multi_fold(list_cons,[([] :- true)|XGs],Rs).

list_cons(X, Xs, [X|Xs]).
```

A different pattern arises from combinatorial search algorithms where one wants to stop multiple threads as soon as a first solution is found. Things like restarts in SAT solvers and various Monte Carlo algorithms fit in this category.

For instance, the predicate `multi_first(K, XGs, Xs)` runs each goal of the form `Xs :- G` on the list `XGs`, until the first `K` answers `Xs` are found (or fewer, if less then `K` answers exist).

It uses a very simple mechanism built into Lean Prolog's multi-threading API: when a `Hub` interactor is stopped, all threads associated to it are notified to terminate.

```
multi_first(K, XGs, Xs) :- hub(Hub),
  length(XGs, ThreadCount),
  launch_logic_threads(XGs, Hub),
  collect_first_results(K, ThreadCount, Hub, Xs),
  stop_interactor(Hub).

collect_first_results(_, 0, _Hub, []).
collect_first_results(0, _, Hub, []) :- stop_interactor(Hub).
collect_first_results(K, ThreadCount, Hub, MoreXs) :-
  K>0, ThreadCount>0,
  ask_interactor(Hub, Answer),
  count_thread_answer(Answer, ThreadCount, ThreadsLeft, Xs, MoreXs),
  ( ThreadCount =:= ThreadsLeft → K1 is K-1
  ; K1 is K
  ), collect_first_results(K1, ThreadsLeft, Hub, Xs).
```

In particular, searching for at most one solution is possible:

```
multi_first(XGs,X) :- multi_first(1,XGs,[X]).
```

The `multi_first/3` and `multi_first/2` predicates provide an alternative to using `CUT` in Prolog as a means to limit search, while supporting a scalable mechanism for concurrent execution. Note also that `multi_first/3` it is more flexible than `CUT` as it can be used to limit the search to a window of `K` solutions. However, in contrast with `CUT`, the order in which these first solutions are found is arbitrary.

5 Agent coordination with cooperative Linda blackboards

The message passing style interaction shown in the previous sections between engines and their clients, can be easily generalized to associative communication

through a unification based blackboard interface [14]. Exploring this concept in depth promises more flexible interaction patterns, as out of order operations become possible, matched by association patterns. An interesting question arises at this point. Can blackboard-based coordination be expressed directly in terms of engines, and as such, can it be seen as independent of a multi-threading API?

We have shown so far that when focusing on performance on multi-core architectures, multi-threading can be encapsulated in high-level constructs that provide its benefits without the need of a complex procedural API.

To support our claim that "concurrency for expressiveness" works quite naturally with coroutining logic engines we will describe here a cooperative implementation of Linda blackboards. In contrast to multi-threaded or multi-process implementations, it ensures atomicity "by design" for various operations. It is an example of *concurrency for expressiveness* that can be used orthogonally from *concurrency for performance* to coordinate cooperatively multiple logic engines within a single thread.

The predicate `new_coordinator(Db)` uses a database parameter `Db` (a synthetic name, if given as a free variable, provided by `db_ensure_bound`) to store the state of the Linda blackboard[5]. The state of the blackboard is described by the dynamic predicates `available/1`, that keeps track of terms posted by `out` operations, `waiting/2`, that collects pending `in` operations waiting for matching terms, and `running/1`, that helps passing control from one engine to the next.

```
new_coordinator(Db) :- db_ensure_bound(Db),
  maplist(db_dynamic(Db), [available/1,waiting/2,running/1]).
```

The predicate `new_task` initializes a new coroutining engine, driven by goal `G`. We shall call such an engine "an agent" in the next paragraphs.

```
new_task(Db, G) :- new_engine(nothing, (G, fail), E),
  db_assertz(Db, running(E)).
```

Three cooperative Linda operations are available to an agent. They are all expressed by returning a specific pattern to the `Coordinator`.

```
coop_in(T) :- return(in(T)), from_engine(X), T=X.

coop_out(T) :- return(out(T)).

coop_all(T, Ts) :- return(all(T, Ts)), from_engine(Ts).
```

The `Coordinator` implements a handler for the patterns returned by the agents as follows:

```
handle_in(Db, T, E) :- db_retract1(Db, available(T)),!,
  to_engine(E, T), db_assertz(Db, running(E)).
```

[5] Note, that, as an extension to standard Prolog, Lean Prolog provides multiple dynamic databases, which, in turn, can be emulated, as shown in [9], in terms of logic engines. Their operations (like `db_assert/2` similar to `assert/1`) have an extra first argument that names the database on which they act).

```
handle_in(Db, T, E) :- db_assertz(Db, waiting(T, E)).

handle_out(Db, T) :- db_retract(Db, waiting(T, InE)),!,
  to_engine(InE, T), db_assertz(Db, running(InE)).
handle_out(Db,T) :- db_assertz(Db, available(T)).

handle_all(Db, T, Ts, E) :-
  findall(T, db_clause(Db, available(T), true), Ts),
  to_engine(E, Ts), db_assertz(Db, running(E)).
```

The `Coordinator`'s dispatch loop `coordinate/1` (failure driven here to run without requiring garbage collection) works as follows:

```
coordinate(Db) :-
  repeat,
    ( db_retract1(Db, running(E)) →
        ask_interactor(E, the(A)), dispatch(A, Db, E), fail
     ; !
    ).
```

Its `dispatch/3` predicate calls the handlers as appropriate.

```
dispatch(in(X), Db, E) :- handle_in(Db, X, E).
dispatch(out(X), Db, E) :- handle_out(Db, X), db_assertz(Db, running(E)).
dispatch(all(T, Ts), Db, E) :- handle_all(Db, T, Ts, E).
dispatch(exception(Ex), _, _) :- throw(Ex).
```

Note also that the predicate `dispatch/3` propagates exceptions - in accordance with a "fail fast" design principle.

```
stop_coordinator(C) :-
  foreach(db_clause(C, running(E), true), stop(E)),
  foreach(db_clause(C, waiting(_, E), true), stop(E)).
```

When the coordinator is stopped using `stop_coordinator`, the database is cleaned of possible records of unfinished tasks in either `running` or `waiting` state. This predicate uses a Lean Prolog extension, `foreach` which makes failure-driven loops more readable - it is defined as follows:

```
foreach(When,Then) :-  When,once(Then),fail.
foreach(_,_).
```

The following test predicate shows that out-of-order `in` and `out` operations are exchanged as expected between engines providing a simple transactional implementation of Linda coordination.

```
test_coordinator :- new_coordinator(C),
  new_task(C, foreach(
      member(I, [0, 2]),
      ( coop_in(a(I, X)),
        write(coop_in=X),write(',')
```

```
  ))),
  new_task(C, foreach(
      member(I, [3, 2, 0, 1]),
      ( write(coop_out=f(I)),write(',')
        coop_out(a(I, f(I)))
  ))),
  new_task(C, foreach(
      member(I, [1, 3]),
      ( coop_in(a(I, X)),
        write(coop_in=X),write(',')
  ))),
  coordinate(C), stop_coordinator(C).
```

When running the code, one can observe that explicit coroutining control exchanges between engines have been replaced by operations of the higher level Linda coordination protocol.

```
?- test_coordinator.
coop_out = f(3),coop_out = f(2),coop_out = f(0),coop_in = f(0),
coop_out = f(1),coop_in = f(2),coop_in = f(1),coop_in = f(3).
```

This shows that "concurrency for expressiveness" in terms of the logic-engines-as-interactors API provides flexible building blocks for the encapsulation of non-trivial high-level concurrency patterns.

6 Coordinating publishers and subscribers

We will now describe a cooperative publish/subscribe mechanism that uses multiple dynamic databases and provides, as an interesting feature, associative search through the set of subscribed events.

The predicate `publish(Channel,Content)` initializes a `Channel`, implemented as a dynamic database together with a time stamping mechanism.

```
publish(Channel,Content) :-
  increment_time_of('$publishing',Channel,T),
  db_assert(Channel,(Content :- published_at(T))).
```

`Content` is a fact to be added to the database, for which the user can (and should) provide indexing declarations to support scalable large volume associative search.

The predicate `consume_new(Subscriber,Channel,Content)` reads the next message on Channel. It ensures, by checking and updating channel and subscriber-specific time stamps, that on each call it gets one *new* event, if available.

```
consume_new(Subscriber,Channel,Content) :- var(Content),
  get_time_of(Channel,Subscriber,T1),
  db_clause(Channel,Content,published_at(TP)),
  T1=<TP, T2 is T1+1,
  set_time_of(Channel,Subscriber,T2).
```

The predicate `peek_at_published(ContentPattern, Matches)` supports associative search for published content, independently of the fact that it has already been read. It provides to an agent the set of subscribed events matching `ContentPattern`.

```
peek_at_published(Channel,ContentPattern, Matches) :-
  findall(ContentPattern, db_clause(Channel,ContentPattern,_),Matches).
```

The predicate `init_publishing(ContentIndexings)` sets up indexing using list of ContentIndexings of the form `pred(I1,I2,...In)` where `I1,I2...In` can be `1` (indicating that an argument should be indexed) or `0`.

```
init_publishing(ContentIndexings) :- index(time_of(1,1,0)),
  maplist(index,ContentIndexings).
```

The following predicates manage the time stamping mechanism, needed to ensure that subscribers get all the events in the order they have been published:

```
set_time_of(Role,Key,T) :- nonvar(Key), remove_time_of(Role,Key),
  db_assert(global_time,time_of(Role,Key,T)).

get_time_of(Role,Key,R) :-
  db_clause(global_time,time_of(Role,Key,T),_), !, T>=0, R=T.
get_time_of(Role,Key,0) :-
  db_assert(global_time,time_of(Role,Key,0)).

increment_time_of(Role,Key,T1) :-
  db_retract1(global_time,time_of(Role,Key,T)), !, T1 is T+1,
  db_assert(global_time,time_of(Role,Key,T1)).
increment_time_of(Role,Key,0) :-
  db_assert(global_time,time_of(Role,Key,0)).

remove_time_of(Role,K) :- db_retractall(global_time,time_of(Role,K,_)).
```

A few other predicates provide clean-up operations, to remove from all the channels the published Content as well as the tracking of subscribers.

```
clean_up_publishing :-
  ( db_clause(global_time,time_of('$publishing',Key,_),_),
    db_clear(Key), fail
  ; true
  ),db_clear(global_time).

clear_channel(Channel) :- db_clear(Channel),
  remove_time_of('$publishing',Channel).

clear_subscriber(Subscriber) :- remove_time_of(_,Subscriber).
```

After defining:

```
pubtest :- init_publishing([wins(1),loses(1)]),
```

```
  maplist(show_goal,[
    publish(sports,wins(rangers)),publish(politics,loses(meg)),
    publish(sports,loses(bills)),publish(sports,wins(cowboys)),
    publish(politics,wins(rand)),
    consume_new(joe,sports,_),consume_new(mary,sports,_),
    consume_new(joe,sports,_),consume_new(joe,politics,_),
    consume_new(joe,politics,_),consume_new(mary,sports,_)
  ]), nl.

show_goal(G) :- G, !, write(G),nl.
```

we can see that the sequence of notifications received by the subscriber agents matches the intended semantics of publish/subscribe:

```
?- pubtest.
publish(sports, wins(rangers)) publish(politics, loses(meg))
publish(sports, loses(bills)) publish(sports, wins(cowboys))
publish(politics, wins(rand))
consume_new(joe, sports, wins(rangers))
consume_new(mary, sports, wins(rangers))
consume_new(joe, sports, loses(bills))
consume_new(joe, politics, loses(meg))
consume_new(joe, politics, wins(rand))
consume_new(mary, sports, loses(bills))
```

The final state of various databases can be queried (and cleaned up) with:

```
?-listings,clean_up_publishing.

% global_time: time_of / 3.
time_of($publishing, sports, 2).
time_of($publishing, politics, 1).
time_of(sports, joe, 2).
time_of(politics, joe, 2).
time_of(sports, mary, 2).

% sports: wins / 1.
wins(rangers) :- published_at(0).
wins(cowboys) :- published_at(2).

% politics: wins / 1.
wins(rand) :- published_at(1).

% politics: loses / 1.
loses(meg) :- published_at(0).

% sports: loses / 1.
loses(bills) :- published_at(1).
```

This shows the presence of sequencing information provided by the dynamic predicate `published_at/1`. Note also that these (indexed) databases can be searched associatively by various subscribers for "past" content.

7 Related work

Multiple logic engines have been present in one form or another in various parallel implementation of logic programming languages, [15, 16] Among the earliest examples of parallel execution mechanisms for Prolog, AND-parallel [7] and OR-parallel [17] execution models are worth mentioning.

However, with the exception of the author's papers on this topic [8, 18, 10, 9] there are relatively few examples of using first-class logic engines as a mechanism to enhance language expressiveness, independently of their use for parallel programming. A notable exception is [19] where such an API is discussed for parallel symbolic languages in general.

In combination with multithreading, our own engine-based API bears similarities with various other Prolog systems, notably [12, 13]. Coroutining has also been present in logic languages to support constraint programming extensions requiring suspending and resuming execution based on changes of the binding state of variables. In contrast to these mechanisms that focus on transparent, fine-grained coroutining, our engine-based mechanisms are coarse-grained and programmer controlled. Our coroutining constructs can be seen, in this context, as focussed on expressing cooperative design patterns that typically involve the use of a procedural multi-threading API.

Finally, our `multi_findall` and `multi_fold` predicates have similarities with design patterns like *ForkJoin* [20] or *MapReduce* [21] coming from sharing a common inspiration source: higher-order constructs like `map` and `fold` in functional programming.

8 Conclusion

We have shown that by decoupling logic engines and threads, programming language constructs for coordination can be kept simple when their purpose is clear – *multi-threading for performance* is separated from *concurrency for expressiveness*. This is achieved via communication between independent language interpreters independent of the multi-threading API.

Our language constructs are particularly well-suited to take advantage of today's multi-core architectures where keeping busy a relatively small number of parallel execution units is all it takes to get predictable performance gains, while reducing the software risks coming from more complex concurrent execution mechanisms designed with massively parallel execution in mind.

The current version of LeanProlog containing the implementation of the constructs discussed in this paper, and related papers describing other aspects of the system are available at *http://logic.cse.unt.edu/tarau/research/LeanProlog* .

Acknowledgment

We thank NSF (research grant 1018172) for support.

References

1. Tarau, P.: Towards Inference and Computation Mobility: The Jinni Experiment. In Dix, J., Furbach, U., eds.: Proceedings of JELIA'98, LNAI 1489, Dagstuhl, Germany, Springer (October 1998) 385–390 invited talk.
2. Tarau, P.: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In: Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents, London, U.K. (1999) 109–123
3. Tarau, P.: Inference and Computation Mobility with Jinni. In Apt, K., Marek, V., Truszczynski, M., eds.: The Logic Programming Paradigm: a 25 Year Perspective, Berlin Heidelberg, Springer (1999) 33–48 ISBN 3-540-65463-1.
4. Tarau, P.: Agent Oriented Logic Programming Constructs in Jinni 2004. In Demoen, B., Lifschitz, V., eds.: Logic Programming, 20-th International Conference, ICLP 2004, Saint-Malo, France, Springer, LNCS 3132 (September 2004) 477–478
5. Mayfield, J., Labrou, Y., Finin, T.W.: Evaluation of KQML as an Agent Communication Language. In Wooldridge, M., Müller, J.P., Tambe, M., eds.: ATAL. Volume 1037 of Lecture Notes in Computer Science., Springer (1995) 347–360
6. FIPA: FIPA 97 specification part 2: Agent communication language (October 1997) Version 2.0.
7. Hermenegildo, M.V.: An abstract machine for restricted AND-parallel execution of logic programs. In: Proceedings on Third international conference on logic programming, New York, NY, USA, Springer-Verlag New York, Inc. (1986) 25–39
8. Tarau, P.: Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In Lloyd, J., ed.: Computational Logic–CL 2000: First International Conference, London, UK (July 2000) LNCS 1861, Springer-Verlag.
9. Tarau, P., Majumdar, A.: Interoperating Logic Engines. In: Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009, Savannah, Georgia, Springer, LNCS 5418 (January 2009) 137–151
10. Tarau, P.: Logic Engines as Interactors. In Garcia de la Banda, M., Pontelli, E., eds.: Logic Programming, 24-th International Conference, ICLP, Udine, Italy, Springer, LNCS (December 2008) 703–707
11. Tarau, P., Boyer, M.: Nonstandard Answers of Elementary Logic Programs. In Jacquet, J., ed.: Constructing Logic Programs. J.Wiley, Hoboken, NJ (1993) 279–300
12. Carro, M., Hermenegildo, M.V.: Concurrency in Prolog Using Threads and a Shared Database. In: ICLP. (1999) 320–334
13. Wielemaker, J.: Native Preemptive Threads in SWI-Prolog. In Palamidessi, C., ed.: ICLP. Volume 2916 of Lecture Notes in Computer Science., Berlin Heidelberg, Springer (2003) 331–345
14. De Bosschere, K., Tarau, P.: Blackboard-based Extensions in Prolog. Software — Practice and Experience **26**(1) (January 1996) 49–69
15. Gupta, G., Pontelli, E., Ali, K.A., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Trans. Program. Lang. Syst. **23**(4) (2001) 472–602
16. Shapiro, E.: The family of concurrent logic programming languages. ACM Comput. Surv. **21**(3) (1989) 413–510
17. Lusk, E., Mudambi, S., Gmbh, E., Overbeek, R.: Applications of the Aurora Parallel Prolog System to Computational Molecular Biology. In: In Proc. of the JICSLP'92 Post-Conference Joint Workshop on Distributed and Parallel Implementations of Logic Programming Systems, Washington DC, MIT Press (1993)

18. Tarau, P., Dahl, V.: High-Level Networking with Mobile Code and First Order AND-Continuations. Theory and Practice of Logic Programming **1**(3) (May 2001) 359–380 Cambridge University Press.
19. Casas, A., Carro, M., Hermenegildo, M.: Towards a high-level implementation of flexible parallelism primitives for symbolic languages. In: PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation, New York, NY, USA, ACM (2007) 93–94
20. Lea, D.: A Java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande. JAVA '00, New York, NY, USA, ACM (2000) 36–43
21. Lämmel, R.: Google's MapReduce programming model revisited. Sci. Comput. Program. **68** (October 2007) 208–237