The Eel Programming Language and Internal Concurrency in Logic Agents

T. S. Dahl *

October 13, 2009

Abstract

This paper describes work done on creating the logic programming language Eel. Eel is an is an extension of prolog which reserves two predicates to handle i/o, process communication and process initiation.

The paper also presents an adaption of a behaviour based agent architecture and gives examples of how parts of that adapted architecture can be implemented in Eel. As an aside the paper comments that the Agent Oriented Programming paradigm currently contains two different metaphors for concurrency.

Eel's event based approach to process communication and process initiation introduces an explicit representation of state to the evaluation of a logic program. The paper demonstrates how Eel can be used for a declarative approach to object states in the examples that show the object oriented implementation of the suggested agent architecture.

1 Introduction

The motivation for this work is the need for a declarative language which can describe common features of agent oriented programming. The long term goal is to have a system where ILP type learning is used to improve existing agent behaviour. In that context it is important to have explicit representations of features like process communication and initiation while keeping the language as simple as possible in order to minimize the search space of any learning algorithm.

Section 2 of this paper presents the Eel programming language, its syntax and its operational and declarative semantics. Eel, is expressive enough to implement complex agent architectures and behaviours while its simplicity also hides as much as possible of the program control from the programmer in the spirit of logic programming.

Section 3 of the papers presents a behaviour based agent architecture which is easier to understand and manipulate than currently existing agent architectures. In particular it makes it easier to manipulate agents that already implement a complex theory of behaviour.

* Machine Learning Research Group, Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol BS8 1UB, United Kingdom. email: tdahl@cs.bris.ac.uk

Sections 4 briefly presents a problem posed as a challenge to autonomous agents for the European Conference on Artificial Life and a solution to that problem based on the Lobster architecture presented in Section 3. Section 5 gives examples of how that solution may be implemented in the Eel language.

Section 6 draws conclusions from the presented work and presents the future directions of this research.

2 The Eel Programming Language

To deal with concurrency, the declarative programming community has developed programming languages with *implicit* [Gre87] as well as *explicit* [PMCA86, Gre97] process communication.

Eel [Dah98] takes the *explicit* approach and is built on a limited model of communication which only allows synchronous point to point communication and which doesn't have the concepts of sending and receiving messages but instead relies on *message unification* as the method of information exchange between processes. This model is chosen for its simplicity and so far, it has been sufficient to describe the agent domain problems considered.

Dedicated send and receive predicates can be implemented on top of this model if this is desireable and the use of dedicated communication processes allows the implementation of asynchronous message passing.

2.1 Eel Syntax

Eel reserves two predicates for handling communication, *event(Event_id,Message)* and *precedes(Event_id1,Event_id2)*. These correspond closely to the Monad type [Wad95] and the temporal sequencing constraint '>>=' used in some functional languages, such as Haskell.

- The *event/2* predicate refers to a communication event. Unless an event matching the description given in the arguments has already taken place and is stored in the *event history*, the evaluation causes the synchronisation of the processes implied by the *Event_id* argument and a unification of the literals specified by the *Message* argument in each of the processes.
- The *precedes/2* predicate refers to two communication events and restricts the second or *constrained* event, to take place only after the first or *constraining* event.
- An *event identifier* has the following format *id(Unique_id,p(Proc_id1,Proc_id2))*. The *Unique_id* argument is a unique identifier for the event, assigned on successful evaluation. The two arguments of the *p* term indicate the two processes involved in the communication. Proc_id1 is called the *transmitting* process name and Proc_id2 is called the *receiving* process name.

  The process naming scheme is inspired by Qu-Prolog [CRH98] in which every process has a unique name which reflects what machine it is running on. The process names *stdin*, *stdout*, *time* and *init* are reserved for user input, user output, communication with a real time clock and the initiation of a new, concurrently running process.

- The *Message* argument can be any literal, but when two synchronising events unify their messages, the result must be ground in order for the event evaluation to succeed. Eel's declarative semantics demands that an unground variable would later have the same instantiation in both processes; something that would involve implicit process communication.

2.2 The Eel Operational Semantics

As in pure Prolog, the operational semantics of all the unreserved predicates in Eel is the process semantics. The reserved predicates have a more complex semantics in order to constrain and synchronise communication.

Global Process Records: During program evaluation, Eel keeps four global records which influence the result of event evaluation:

- **H** is the *event history*. It records all the *historic events*, events that a process has successfully evaluated. The *unique identifiers* of the events record the order in which they were evaluated.
- **Ws** records all the disjuncts in the current goal that are currently suspended awaiting synchronisation with other processes.
- **Wc** records disjuncts awaiting the satisfaction of their *constraints*. When a *constrained event* is evaluated, the disjunct it is a part of is recorded here and further evaluation is delayed until the *constraint* is satisfied.
- **P** records external processes that have requested event synchronisation with this process.

Records Local to Disjuncts: In addition to the global process records records, Eel keeps two local records for each disjunct in the current goal

- **UC** records all the *unsatisfied constraints* that have been evaluated in this branch of the SLD-tree. A constraint is unsatisfied if there is no *historic event* matching the *constraining event identifier*. When an event matching the *constraining event identifier* is evaluated, the now *satisfied constraint* is removed from this record.
- **SC** records *satisfied constraints* for which there are no *historic events* matching the *constrained event identifier*. When an event matching the *constrained event identifier* is evaluated, the constraint is removed from this record.

In order to satisfy the declarative semantics of Eel, there must be *historic events* to match both the *constraining* and the *constrained event identifiers*. The evaluation of a disjunct fails if, at the end, the local records **UC** and **SC** contain any constraints.

The evaluation of an *event(Event_id,Message)* literal: The evaluation of an event literal follows the steps given below:

1. The *event identifier* is checked for variables. If the *unique event identifier* is a non-ground variable, a new disjunct is created for every *historic event* which matches the event. In addition a disjunct is created in which the event is given a *unique event identifier* which indicates that the event is a *future event*

2. If the *unique event identifier* is ground, **H** is checked to see if the event literal can be unified with a *historic event*. If it can, the evaluation succeeds.

3. If the ground *unique event identifier* indicates that the event is a *future event*, **UC** is checked for *unsatisfied constraints* on the event. If there are any such constraints, the disjunct containing the event is added to **Wc**.

4. If there are no *unsatisfied constraints* on the event, the *process names* are checked to see if they indicate an *internal*, *external* or *special* event. Internal event are indicated by having the evaluating process' name in both the *transmitting* and then *receiving* process name fields. An *internal* event immediately succeeds and is added to **H**. Such events can be used to keep track of the evaluation state of a single process without refering to external processes.

5. If the event is *special*, it indicates either user input, user output, a real time check or the initiation of a new process. A user input event succeeds if the input unifies with the event *message*, a user output event succeds immediately on printing the event *message*, a real time check event succeeds by unifying the event *message* with a representation of the current time, and a process initiation event succeeds if a new process can be initiated with the process name given in a structured *process initiation message*. The query part of the *process initiation message* is treated as the initial query in the new process. The *process initiation message* has the structure $(New\_proc\_name, Query)$.

6. If the event is *external*, **P** is checked for an event synchronisation request from the *receiving* process. If no such request is registered, Eel requests synchronisation for this event from the other process and the disjunct containing the event is recorded in **Ws**.

7. If the other process is registered in **P**, a *synchronisation point* has been found. Eel then interrupts the other process and attempts to unify the message from the other process with the message in this process. If the unification fails or the result is not ground, the evaluation of the event fails.

When an event evaluation succeeds, the event is added to **H**. The other process' synchronisation request is removed from **P**. All *constraints* in **UC** which have the evaluated event as the *constraining* event are removed and all the constraints in **SC** which have the evaluated event as the *constrained* event are removed. If any recorded constraint contains a variable for either the *constraining* or the *constrained* event, new disjuncts are added to the goal with these constraints unified with the evaluated event.

Finally, if any disjuncts from **Wc** are now available to be evaluated, they are added to the current goal.

When evaluation is interrupted by another process that has found a *synchronisation point*, and a successful ground unification of the messages is made, any disjunct containing the successfully evaluated event is added to the front of the current goal. The disjunct is then removed from **Ws** and the evaluation of the event on which the disjunct was supspended succeeds.

So far, a detailed operational semantics has not been developed for variables in the process names, but the intuitive interpretation is to then create a disjunct for all communicating processes that can ground the given variables. This would allow an event with a variable in the *receiving* process id to communicate with any process, including itself, i.e. interpret the event as *internal*.

The evaluation of a *precedes(Event_id1,Event_id2)* literal: The evaluation of a *precedes(Event_id1,Event_id2)* literal also called a *constraint*, follows the sequence given below:

1. The constraint's event identifiers are checked for variables. If there are any variables in the *unique event identifiers*, disjuncts are added to the current goal from all the combinations of *historic events* that satisfy the constraint. In addition the constraint is added to the *constraint records* **UC** and **SC** of the disjunction currently being evaluated.

2. If the constraint's *unique event identifiers* are ground, **H** is checked for events that have *event identifiers* which unify with the two *event identifiers* given in the *constraint*. If two events from **H** can be found which satisfy the *constraint*, the evaluation succeeds.

3. If an event matching the *constrained event* is found, but no match is found for the *constraining event*, the evaluation fails.

4. If an event matching the *constraining event* is found, but no match is found for the *constrained event*, the evaluation succeeds.

5. If no event is found to match any of the events specified, Eel records the constraint in the *constraint records* and the evaluation continues.

2.3 The Eel Declarative Semantics

As in Prolog, the program **P**, is a set of horn clauses. Evaluation is initiated by a query **q**, and the declarative semantics are that all the variable-bindings θ, which make the program imply the query, are found.

Eel abduces the reserved literals needed to deduce **q** from **P**. The abduced *constraints* form a partial ordering R_E, on the set of abduced *event literals*, **E**.

Two processes communicating with eachother have different for the *transmitting* and *receiving* processes. To give a unique declarative representation of the corresponding communication event, the declarative interpretation of the process name part of an *event identifier*, $p(Transmitting\_proc, Receiving\_proc)$, is an unordered set of two process names.

The declarative semantics of Eel states that **E** is abduced in such a way that the partial ordering described by R_E is a subset of the temporal ordering T_E on the abduction time of the *event literals*.

In plain words, the order in which the events take place or are abduced, respects the ordering described by the abduced *constraints*.

Formally this can be expressed as:

- $(P \wedge E \wedge R_E \rightarrow q\theta) \wedge R_E \subseteq T_E$

With respect to the declarative semantics, the operational semantics of Eel are sound but not complete. The incompleteness results from the commitment to a restricted set of temporal orderings which is a result of the arrangement that the abduction of an *event literal* corresponds to doing input or output.

It is also important to point out that the set of abduced events is a subset of all the events successfully evaluated.

Process Initiation: To make the declarative semantics of process initiation comply with the operational semantics, the precence of the clause

$event(id(\_, p(\_, init)), (New\_proc\_name, Query) \rightarrow Query$

is always assumed in the declarative interpretation of an Eel program. This clause is called the *process initiation clause*. Its precence forces the logical interpretation of an Eel program to coincide with the result of the evaluation by demanding that the event *query* is true whenever a event literal which indicates process initiation is abduced.

3 Concurrent Agent Architectures

There are indications that an evolutionary inspired design method based on building complex behaviours as small additions to a concurrent set of simpler behaviours is the best way of designing agents with efficient, robust and complex behaviours. This behaviour based approach has its roots in the Subsumption Architecture [Bro91] which was very successful in controlling basic robot behaviour, but which turned out however to create a lot of problems for system developers in that it merged the flow of data with the flow of control.

Two Metaphors for Concurrency: In 1993, Shoham coined the term *Agent Oriented Programming* [Sho93] and presented a traditional sense-act cycle as a way of dealing with mental modalities like commitment and intentions. That and other work on representing such modalities shifted the focus of agent research away from communication between concurrent processes towards the internal processes of an agent.

A few years before, Brooks had criticised such architectures for their lack of concurrency and for their explicit representation of mental properties [Bro91]. Brooks instead suggested a fully concurrent Subsumption Architecture. The Subsumption Architecture gave rise to the Behaviour Based Approach to AI (BBAI), but as BBAI matured, more hierarchically structured architectures were suggested, with support from research in biology [BM97]. The new BBAI architectures have greatly improved in comprehensibility, while building on the strengths of the Subsumption Architecture, concurrency and inspiration from evolution.

Brooks' criticism brought agent research back to its origin, the question of concurrency. As a result, AOP now contains two metaphors for concurrency; the traditional view of concurrent programs as communicating agents, and the novel behaviour based view of concurrent programs as interacting behaviours.

3.1 The Lobster Architecture

A particular BBAI agent architecture that has been successfully applied to both robot control and to a simulated environment, is the Edmund architecture [BM97]. The Lobster architecture builds on the Edmund architecture, with a few important differences.

Both architectures have a set of concurrent *drives* which again consist of a number of concurrent *competences*. The *competences* also consist of further *competences* or of *fixed action patterns*. *Fixed action patterns* are sequences of bottom level atomic *actions*.

A *drive* has a given *priority* that can override or be overridden by other *drives' priorities*. On startup, the *drives* register *triggers* with different *senses*. *Senses* are concurrently running processes that create sensible data from the bottom level *sensors*. During runtime, the *senses* trigger the *drives*. The drives initiate their *competences* which again activate further *competences* or *fixed action patterns*. An initiated *fixed action patterns* executes uninterrupted.

The main difference between the Edmund and the Lobster architectures is that Lobster *drives* run truly concurrently. This forces some arbitration between the desired *actions* of different *drives* in order to respect *fixed action patterns* and avoid conflicting *actions*. The Lobster architecture introduces an *arbitration module* as a common interface to the *actuators*. This module introduces further structure to the agent and is another step away from the strict concurrency demanded in the Subsumption Architecture. The reason for introducing it against the behaviour based tradition is the increased comprehensibility it introduces over the alternative of a network of inhibition and excitation channels between *actuators*. In addition there is evidence that the basal ganglia, a modularised part of the brain, serves a similar purpose in animals [GPR98].

4 An Example Behaviour Based Solution

To give an example of the functionality of the Lobster architecture and the Eel language, Sections 5.1 and 5.2 show examples from the implementation of an agent solving a simple problem presented for the Artificial Life Creator Contest which took place during the Fifth European Conference on Artificial Life, ECAL'99.

The Food Race Problem: The given task is to control an agent which is a simulated version of a Khepera robot. The agent competes with one other agent and has to navigate a random environment to find *feeders*. When a feeder is found the agent has to position itself correctly so that it receives the energy from the feeder. Over time the

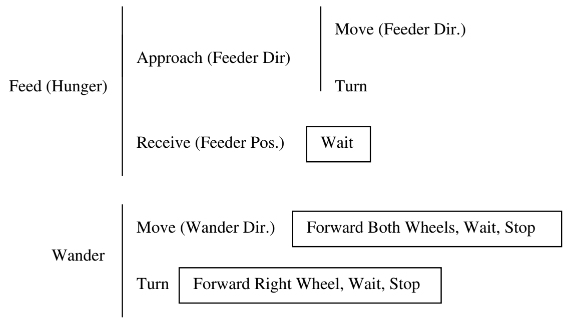

Figure 1: The Wander and Feed Solution

agent uses energy and the feeders are refilled at a decreasing rate. If an agent goes too long without feeding it dies, and the agent that is the last one alive is the winner.

The agents have a simulated infrared distance sensor and a simulated 80x60 pixels colour camera as their given sensors.

The world contains a maze with a number of feeders and a number of landmarks which can be used to build maps for remembering the positions of the feeders.

The Wander and Feed Solution: The agent designed to solve this problem has two *drives*; *feed* and *wander*. The *wander drive* keeps the agent moving and the *feed drive* overrides it whenever a feeder is observed. The solution is visualised in Figure 1.

The Lobster architecture is very naturally implemented as objects. Three *senses* are implemented to run concurrently on top of the *sensors*. The *senses* are; *direction of present feeder*, *feeding placement*, and for simplicity, *most promising wander direction*.

The *feed drive* initiates the two *competences*; *approach* and *receive*. If a feeder appears, the *priority* of the *approach competence* will send the agent towards it. When the *feeding placement sense* signals that the agent is close to the feeder, the *receive competence* overrides the *approach competence* and positions the agent correctly for receiving food.

The *wander drive* initiates two *competences*; *turn* and *move*. If no *wander direction* is available the *turn competence* will turn the agent. If a *wander direction* becomes available, the *move competence* will override the *turn competence* and make the agent move forward.

This is a simplified solution to a simplified problem, but it describes how a behaviour based problem would be solved in the Lobster architecture.

5 Implementing the Lobster architecture in Eel

The examples provided in this section are meant to give an idea of how Eel code looks and how the reserved predicates are used to solve problems of process communication and initiation.

5.1 Object States in Eel

Eel's *event history* indirectly represents an object's state. This is exemplified here by the *arbitrator module* from the Lobster architecture, here implemented as an object. When a *fixed action pattern (fap)* requests *actions*, the *arbitrator module* must allow it to finish uninterrupted. To do this it enters a *fap exclusively* state where it only accepts events from the *fap* until the *fap* process sends a *done* message. Pure Prolog is not able to handle object state, and this has been considered the reason for the failing merger of the LP and the OOP paradigms. Eel now makes it possible to implement all aspects of OOP in a declarative logic language.

The code for checking that a message has previously been received from a *fap* and that the last message received from the *fap* was not a *done* message, is given below. This is the test used by the *arbitrator module* to find out if it is in a *fap exclusively* state. The *time event* is an *internal event* which is used to create a temporal reference point for the current state.

```
accept_action_requests:-
   fap_exclusively_state,
   event(id(Fid,p(arb,fap)),Message)
   handle_message(Message),
   accept_action_requests.
accept_action_requests:-
   not(fap_exclusively_state),
   event(id(Id,p(arb,Any_proc)),Message),
   handle_message(Message),
   accept_action_requests.

fap_exclusively_state:-
   event(id(Int_id,p(arb,arb)),true),
   precedes(id(Fid1,p(fap,arb)),id(Int_id,p(arb,arb))),
   event(id(Fid1,p(fap,arb)),Message),
   not(precedes(id(Fid2,p(arb,fap)),id(Int_id,p(arb,arb))),
       precedes(id(Fid1,p(arb,fap)),id(Fid2,p(arb,fap))),
       event(id(Fid2,p(arb,fap)),Message),
       Message!=done).
```

5.2 Concurrent Evaluation in Eel

For concurrent evaluation, Eel can spawn processes. One example is a *drive* that s-pawns several *competences* and passes them the identities of the *senses* so that they can register their *triggers* with these.

The reserved process identity *init*, indicates that a new process is to be initiated. In this case the message holds the new process name and the initial query. The event only succeeds if Eel manages to initiate a new process with the given name evaluating the given query.

In the code below the *feed drive* spawns its *competences* when it gets hungry and shuts them down again when it is full.

```
feed(Hunger_need,Feeder_sense,Pos_sense,Arb):-
   passive_feed(Hunger_need,Feeder_sense,Pos_sense,Arb).

passive_feed(Hunger_need,Feeder_sense,Pos_sense,Arb):-
   event(id(Id,p(Hunger_need,feed)),Hunger),
   init_comps(Hunger,Hunger_need,Feed_sense,Pos_sense,Arb).

init_comps(Hunger,Hunger_need,Feeder_sense,Pos_sense,Arb):-
   Hunger>5,
   event(id(Id1,p(feed,init)),(app,app(Feeder_sense,Arb))),
   event(id(Id2,p(feed,init)),(rec,rec(Pos_sense,Arb))),
   active_feed(Hunger_need,Feeder_sense,Pos_sense,Arb).
init_comps(Hunger,Hunger_need,Feeder_sense,Pos_sense,Arb):-
   passive_feed(Hunger_need,Feeder_sense,Pos_sense,Arb).

active_feed(Hunger_need,Feeder_sense,Pos_sense,Arb):-
   event(id(Id,p(Hunger_need,feed)),Hunger),
   shut_comps(Hunger,Hunger_need,Feeder_sense,Pos_sense,Arb).

shut_comps(Hunger,Hunger_need,Feeder_sense,Pos_sense,Arb):-
   Hunger<=5,
   event(id(Id1,p(feed,app)),shutdown),
   event(ids(Id,p(feed,rec)),shutdown),
   passive_feed(Hunger_need,Feeder_sense,Pos_sense,Arb).
shut_comps(Hunger,Hunger_need,Feeder_sense,Pos_sense,Arb):-
   active_feed(Hunger_need,Feeder_sense,Pos_sense,Arb).
```

6 Conclusions and Future Work

The Eel language is hoped to be a simple and clean, but still intuitive approach to concurrent and distributed logic programming. It has not been lacking in expressive power when it has been used to design solutions to complex problems like the food race presented in Section 4.

The results presented here are mainly from work done on an initial prototype Eel interpreter.

An improved Eel interpreter is currently under development, using Sicstus Prolog and Linda, this work continues to bring to light new issues that follow from this approach to concurrent and distributed logic programming.

The Eel framework also needs further developing, particularily with respect to allowing unground variables in the process names of an *event identifier* and with regards to programming with negation.

References

[BM97] J. J. Bryson and B. McGonigle. Agent Architecture as Object Oriented Design. In *Intelligent Agents IV, Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL'97), LNAI 1365*, pages 15–30. Springer Verlag, 1997.

[Bro91] R. A. Brooks. Intelligence without reason. In *Proceedings of IJCAI 91*, pages 569–595. Morgan Kaufmann, 1991.

[CRH98] K. Clark, P. Robinson, and R. Hagen. Programming internet distributed DAI applications in Qu-Prolog. Presented at the Australian DAI WS, Brisbane, 1998.

[Dah98] T. S. Dahl. *Eel*, A Declarative Language for Programming Adaptive Agents. Technical Report CSTR-98-008, Department of Computer Science, Bristol University, August 1998.

[GPR98] K. N. Gurney, T. J. Prescott, and P. Redgrave. The Basal Ganglia viewed as an Action Selection Device. In *Proceedings of the Eighth International conference on Artificial Neural Networks*, 1998.

[Gre87] S. Gregory. *Parallel Logic Programming in PARLOG, The Language and its Implementation*. International Series in Logic Programming. Addison-Wesley, 1987.

[Gre97] S. Gregory. A Declarative Approach to Concurrent Programming. In H. Glaser, editor, *Proceedings of the 9th International Symposium on Programming Languages, Implementations, Logics, and Programs*. Springer-Verlag, 1997.

[PMCA86] L. M. Pereira, L. Monteiro, J. Cunha, and J. N. Aparicio. Delta prolog: A distributed backtracking extension with events. In E. Y. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, volume 225 of *Lecture Notes on Computer Science*, pages 69–83. Springer-Verlag, 1986.

[Sho93] Y. Shoham. Agent-oriented programming. *AI Journal*, 60(1):51–92, 1993.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer-Verlag, 1995.