Eel, a Declarative Language for Programming Adaptive Agents

T. S. Dahl

August 1998 **CSTR-98-008**

University of Bristol

Department of Computer Science

Also issued as ACRC-98:CS-008

Abstract

This is a report on the *Eel* programming language, its syntax, semantics and intended use. The language has been developed as a tool to allow programs to adapt using symbolic learning techniques. *Eel* is a logic language and has a declarative interpretation of user and process communication. This is achieved by extending the traditional deductive proof procedure with abduction of a set of communication events.

Eel also allows initiation of concurrent evaluation of subqueries and has a declarative interpretation of such initiations.

1 Introduction

Adaptive agents with a declarative representation of the rules that guide their behaviour is an important extension to today's adaptive agents. It is currently common for adaptive agents to use non-declarative methods of adaption, like artificial neural networks.

An agent with a declarative representation of its rules of behaviour can use symbolic learning methods like Inductive Logic Programming (ILP) [9], to synthesise more such rules. Such rules, in the form of programs can be synthesised from examples provided by interaction with other external processes.

Another advantage of having rules with a declarative representation is that this rules can be easily understood by humans. This gives a better possibility for guided adaption and allows an agent's user to better understand its behaviour.

A declarative approach to program synthesis and agent adaption demands a declarative interpretation of process interaction.

The *Eel* language provides a declarative framework which allow agents to adapt using ILP techniqes. The main idea of *Eel* is to handle all interaction between processes and users with the same abstraction, that of communication events. In addition, *Eel* extends the traditional, deductive Prolog proof procedure with abduction of terms formed using a set of reserved predicates.

The idea of making process communication in Prolog explicit was introduced in Delta Prolog [11]. The idea of using events to synchronize concurrent processes and perform communication was also introduced in the Tempo language [4], where they are also used to reason about a systems safety properties.

The **IFF** proof procedure [3] expanded the classical deductive proof proocedure with abduction. The Openlog language [1] was built on that proof procedure.

2 The *Eel* Syntax

2.1 Reserved Predicates

The syntax of the *Eel* language is based on the Prolog syntax [7], but reserves two predicates, *event(Event_id,Message* and *precedes(Event_id1,Event_id2)* for user and process communication.

- The *event/2* predicate refers to a communication event and if that event has not yet taken place, the evaluation causes the synchronisation of the involved processes and the unification of the involved messages.

- The *precedes/2* predicate refers to two events and restricts the second to take place only after the first one has taken place. The event that is specified by the event identifier in the first argument of the predicate is called the *constraining* event. The event that is refered to by the identifier in the second argument of the predicate is called the *constrained* event. A literal containing the *precedes* predicate is called a *precedence constraint*. If the second event has not yet taken place when a litral containing the *precedes* predicate is evaluated, the constraint is recorded and respected when evaluating future events.

2.2 Reserved Terms

Eel also reserves a set of terms for description of channels in the *event identifiers*.

- An *event identifier* has the form of a tuple **(Channel_id,Message_no)**. The *channel identifier* specifies what channel the communication is to take place on and the *message number* is used to differentiate between and order different messages on the same channel. The *message* has the form of a clause.

The channel identifiers *std_in*, *std_out* and *(i,N)*, where N is any integer, are reserved for user input, user output and initiation of concurrent evaluation respectively.

Two messages with the same *channel identifier* are also constrained as to when they can take place. The event with the lower *message number* must take place first. Thus, the message numbering has the effect of explicitly constraining the event precedence, in exactly the same way as the *precedes* predicate and is similiaily included in the set of *precedence constraints*.

2.3 Ordering of Literals

The result of the operational semantics of the reserved predicates is that the operational semantics of *Eel* is not complete. When a communication event involving an external process or user has been executed it is not possible to undo it. This gives the order in which literals appear in the code an influence on the result of the evaluation. Below is an example of how two equivalent logic sentences will give different answers to a query. The intended functionality is to prompt a user for his/her name with an output event and then to read that name with an input event.

```
interact1:-
     event((std_out,1),'Please enter your name: '),
     event((std_in,1),Name).

interact2:-
     event((std_in,1),Name),
     event((std_out,1),'Please enter your name: ').
```

The *interact1* clause will prompt the user before accepting an answer, while the *interact2* clause will read a name from the user before writing the prompt.

The ordering of the *event* literals influences the outcome of the evaluation together with the *precedence constraints*. The *precedence constraints* can only be used to exclude undesired solutions. It can not be used to guarantee the desired ones. This is why they are explicit declarations of the *safety properties* of the program. They are not specifications of the *progress properties*.

Another program that behaves like *event1* and one that will fail are given below in *interact3* and *interact4* respectively.

```
interact3:-
     precedes((std_out,1),(std_in,1)),
```

```
      event((std_in,1),Name),
      event((std_out,1),'Please enter your name: ').

interact4:-
      event((std_in,1),Name),
      precedes((std_out,1),(std_in,1)),
      event((std_out,1),'Please enter your name: ').
```

It may seem cumbersome to explicitly specify the precedence constraint when the two clauses *interact1* and *interact3* will give exactly the same answers and *interact4* will fail. This seems to imply that the explicit precedence constraints do not add any functionality to the language and could with good reason be made implicit in the ordering of the literals.

The counter example below shows a program ensuring mutual exclusion to a resource accessible throung two channels, utilising the functionality of explicit precedence constraints.

```
mutex((E1_ch,E1_no),(E2_ch,E2_no)):-

      event((E1_ch,E1_no),reserve),
      precedes((E1_ch,E1_no+1),(E2_ch,E2_no)),
      mutex((E1_ch,E1_no+1),(E2_ch,E2_no));

      event((E2_ch,E2_no),reserve),
      precedes((E2_ch,E2_no+1),(E1_ch,E1_no)),
      mutex((E1_ch,E1_no),(E2_ch,E2_no+1));

      event((E1_ch,E1_no),release),
      mutex((E1_ch,E1_no+1),(E2_ch,E2_no));

      event((E2_ch,E2_no),release),
      mutex((E1_ch,E1_no),(E2_ch,E2_no+1)).
```

Another important reason to keep the precedence constraints explicit is that in *Eel* the state of the program evaluation can be found by analysing previous events. The implementation of the *holds_at* clause from the event calculus [6] given below shows how the *precedes* predicate is essential such analysis.

```
holds_at(Relation,Last_event):-
      initiates(Previous_event,Relation),
      precedes(Previous_event,Last_event),
      Event,
      not(broken(Relation,Previous_event,Last_event)).
```

The examples above assume a computation rule that resolves on the leftmost literal, like in Prolog and also in *Eel*. This means that an event that is to precede another event must have its leftmost appearance placed syntactically to the left of the leftmost appearance of the second event and that only precedence constraints placed to the left of the event they constrain will influence the ordering in which the events take place.

3 The *Eel* Operational Semantics

Like Prolog, the operational semantics of all the unreserved predicates in *Eel* is the process semantics [7]. The reserved predicates have a more complex semantics.

3.1 *Eel* Process Records

Eel processes keep four records. These records are the external communication channels **Ch**, the event history **H**, the current constraints **Co**, the events delayed awaiting the satisfaction of constraints **HC**, and the processes awaiting communication with the process in question **P**.

1. **Ch** records what channels are external to that process. The collection of all these records from the existing processes reflects the hardware stucture the system is runing on. This structure must be declared before the initial query. All processes have the external communication channels *std_in* and *std_out*.

   All channels to processes initiated using the **(i,N)** event identifier are shared between the initiating and the initiated processes only. These channels are added to the external channel record at initiation time. Processes initiated in this way run on the same processor as the initiating process.

2. **H** records what events have taken place and the order in which they took place.

3. **Co** records unsatisfied constraints which concern events that have not yet taken place. As constraints are satisfied, they are removed from this record.

4. **HC** records what events are delayed waiting for a constraint to be satisfied. When the constraints are satisfied the events recorded here immediately take place. If this record is not empty by the end of the evaluation, the evaluation fails.

5. **P** records what other processes are awaiting communication with the process. A process is added to this record in another process when it tries to evaluate an event on a communication channel that it shares a with that other process.

All these records are kept so that it is possible to construct a set of events that together with the program satisfies the initial query.

3.2 Reserved Predicate Evaluation

The *event(Id,Message)* predicate The following happens when a literal containing the *event* predicate is evaluated.

1. It is first checked with the history of events **H**, if that event has alread taken place. If that is the case, the evaluation of the literal succeeds or fails according to the unifyability of the recorded message and the message specified in the literal being evaluated.

2. If the event has not yet taken place, it is checked whether there are currently any unsatisfied constraints concerning that event in **Co**. If there is then that event is added to the record of halted events awaiting the satisfaction of a constraint **HC** and the evaluation is delayed.

3. If there are no unsatisfied constraints concerning the event, the redcord of external channels **Ch**, is checked to see if the the event is an *external event*, that is, the event takes place on a channel that is shared with another process. If the event is not an *external event*, it immediately succeeds and is added to the record of events that have taken place.

4. If the event is an external event, the record of waiting processes **P**, is checked to see if all the other involved processes are ready. The user input and user output processes are registred as always ready. If they are not ready the disjunct in which the literal appears is suspended and the process is entered as a waiting process on all the other processes' records of waiting processes.

5. If all the involved processes are ready, the different *event messages* are unified. If this unification fails then the evaluation of this event fails in all the processes involved.

When the evaluation of an event succeds the following happens.

1. The event is recorded in all the processes' records of events that have taken place, **H** with the unified *event message*.

2. All the processes recorded as waiting for the succeedded event in **P** are removed.

3. The constraints on the record of not satisfied constraints **Co** that have the succeeded event as the *constraining* event are removed.

4. All the events on the record of events delayed awaiting the satisfaction of a constraint **HC** that are waiting for the satisfaction of constraints that have the succeeded event as the *constraining* event are evaluated.

When the evaluation of a literal containing the *event* predicate fails, the evaluation of the *Eel* program fails.

The *precedes(Id1,Id2)* predicate When a literal that contains the *precedes* predicate is evaluated the following happens.

1. It is first checked with the history of events **H**, if the *constrained* and the *constraining* event has already taken place.

2. If the *constrained* event has taken place but the *constraining* event hasn't, the evaluation fails.

3. If both the *constrained* and the *constraining* event have taken place, it is checked with the event history **H** if the constraint is satisfied and the evaluation succeeds or fails according to it being satisfied or not.

4. If the *constrained* event has not taken place and the *constraining* event has, the evaluation succeedes.

5. If none of the events have taken place, the constraint is recorded as a not satisfied constraint on an event that has not yet taken place on **Co** and the evaluation succeeds.

3.3 An Alternative Semantics for *Eel*

It would be possible to change the operational semantics of *Eel* so that the clause *interact4* given above in Section 2.3 would succeed, but this would mean restricting some of the meta level programming possibilities that make Prolog so powerful. In particular it would not be possible to use an argument from a literal as a new literal in the way demonstrated by the *meta_demo* program given below.

```
meta_demo:-
     event(std_out,'Please enter mode [true/false]: '),
     event(std_in,Mode),
     possible(precedes((std_in,1),(std_out,1)).

possible(true,Constraint):-
     Constraint.
possible(false,_).
```

It is impossible to tell in advance in what order the events should take place, and when the events have been executed, the constraint is already violated. It is considered most important to preserve the interactive facilities in the *Eel* language, so the rule that only constraints that are placed to the left of the events they constrain can influence the event ordering, is left standing.

4 The *Eel* Declarative Semantics

As in classic Prolog there is a program **P**, which is a set of sentences in first order predicate logic in the form of horn clauses. Execution is initiated by a query **q**, and the declarative semantics of the program execution is that all the bindings, θ which make the program imply the query, are found for variables in the query.

The main difference introduced with events is that a set of communication events **(E,R)** is constructed, with **E** being a set of event clauses and **R** being a set of precedence constraints. The partial order introduced by **R** is the temporal precedence introduced by the time the individual events take place. The declarative semantics for the program

P and the query **q** is that, a set of events, $(\mathbf{E}, \mathbf{R})$, is constructed in addition to the classic variable bindings θ, for the query **q**. The set of events, **(E,R)** together with the variable binding θ and the program **P** implies the query, **q**.

- $P \wedge E \wedge R \rightarrow q\theta$

4.1 Concurrent Evaluation

When concurrent evaluation is initiated, the initiation takes the form of a communication event with a query as its value. In the initial query, it is possible to represent several processes which can participate in such communication events.

It is also possible to spawn new processes through the use of reserved *channel identifiers*. The set of *channel identifiers (i,N),* where *N* is any integer has been reserved for spawning new processes. The initial query implicitly contains a representtion of a set of processes that await communication on these channels.

The explicit representation of one of the processes implicit in the query is given below.

```
process(Id):-
      event(Id,Query),
      Query.
```

5 Proving Properties of *Eel* Programs

The safety properties of an *Eel* program are explicitly stated in its *constraints* on external events. For a given set of external events the evaluation can be proved to never reach certain states. The evaluation can be proved to never reach certain states in general by doing s aimmilar proof for an exhaustive set of external events.

This technique is based on the work done on explicit safety constraints in the **Tempo** language [4].

6 Combining *Eel* Programs

Two disjuctive subsections of an *Eel* program can contain references to different sets of internal events. If a unique event is refered to in both of the two subparts, a particular phenomena occurs. If the two subparts are evaluated concurrently this event will cause synchronisation and communication. If they are evaluated sequentially the first event automatically succeed and later evaluations of that event will refere to the recorded event.

In both cases the event evaluation is backtrackable and the event evaluation will not lead to incompleteness.

7 The *Eel* Deductive/Abductive Proof Procedure

The *Eel* operational semantics can be described by a proof procedure with both deductive and also some particular abductive inference rules. The deductive inference rules given in Equation 1 and Equation 2 are the same as for Prolog but with the added symbol **D** describing the set of abduced literals. **T** is the theory on which the proof procedure operates.

$$\frac{T \cup D, a \in T}{T \cup true \cup D} \tag{1}$$

$$\frac{T \cup D, \{b, a \leftarrow b\} \in T}{T \cup a \cup D} \tag{2}$$

Combining reasoning with defined and undefined predicates in the same proof procedure has been the focus of abductive logic programming. The IFF proof procedure [3] defines a program as a tuple **¡T, IC, Ab¿**. **T** is a set of definitions of predicates in *iff* form rather than the classical *if* form of Prolog. **IC** is a set of integrity constraints and **Ab** is a set of undefined but abducible predicate symbols.

Given an abductive logic program, an answer is a pair (D, σ), where **D** is a set od ground abduced atoms and σ is a substitution for the variables in a query **Q**. The two following relations hold for abductive programs.

- $T \cup Comp(D) \cup CET \models Q\sigma$
- $T \cup Comp(D) \cup CET \models IC$

Comp(D) is the completion of D in *iff* form and CET is the CLark Equality Theory. The **IFF** proof procedure is both sound and complete.

The *Eel* language is described below as an abductive program with only the *event* predicates and the *precedes* predicate as abducables.

- **T** = **P**
- **Ab** = $\{$ event/1, event/2, precedes/2 $\}$
- **IC** =
  - $Id1 = initial \vee$
    $event(Id1, Msg1) \wedge precedes(Id1, Id2) \leftarrow event(Id2, Msg2),$
  - $false \leftarrow precedes(Id1, Id2), precedes(Id2, Id1)$

The first implication in the integrity constraints says that all events other than the initial one must be preceded by another event. The second implication says that two events cannot precede eachother.

Given such a program the IFF proof procedure will produce a set of events and an ordering on them which

The *Eel* operational semantics will only abduce one particular set of events satisfying the constraints given in the program and might not even succeed in doing that. More work will be done on connecting the *Eel* operational semantics to a formal proff procedure in order to maximise the possibility of producing a solution to any given query.

8 Using *Eel* for Programming Adaptive Agents

This section describes a typical application for the *Eel* language. The example is very simple, but serves to describe different general issues in programming anadaptive agent.

8.1 An Example Adaptive Agent

A basic 'perceive-act' cycle has been implemented as a recursive clause. It runs concurrently with a boolean object with which it communicates.

From the set of communication events an ILP algorithm will synthesise definitions for the different possible events. These definitions will again be used as extensions to the program, effectively changing its behaviour according to what it has learnt.

An example agent loop is given below. The abstraction of *goals* and *beliefs* are taken from [8] and [12] respectively.

```
l_agent(Bool_event_id):-
     fulfilled(Bool_event_id,[],[]).

fulfilled(Event_id,Beliefs):-
     goal(Goal),
     actions(Event_id,Goal,Beliefs,Actions),
     act(Actions,Next_event_id),
     learn(Next_event_id,Beliefs,New_beliefs),
     fulfilled(Next_event_id,New_beliefs).
```

A concurrently evaluated boolean object accepts *value(Value)*, *set* and *reset* events, where every second message must be a *value(Value)* message. Until the agent has learnt how to manipulate the boolean object, its primary goal is to expand its experience by trying events in new contexts. After having tried all possible events in all available contexts the agent has the following experience or *history*, $\mathcal{H}$.

- $\mathcal{H}$ =

```
{event(((i,1),1),value(true)),
 event(((i,1),2),set),
 event(((i,1),3),value(true)),
 event(((i,1),4),reset),
 event(((i,1),5),value(false)),
 event(((i,1),6),set),
 event(((i,1),7),value(true)),
 event(((i,1),8),value(true))}
```

8.2 Synthesising *Eel* Programs

It is a goal for further work to find a way in which the agent can generalise definitions of the different events from experimental data. The desired definition is given below.

The relation is renamed *learnt_event(ID,Value)* so as to not redefine the *event(Id,Value)* predicate.

- Σ =

```
learnt_event((i,1,N+1),set):-
     event(((i,1),N),value(_)),
     event((i,1,N+1),set).
learnt_event((i,1,N+1),reset):-
     event(((i,1),N),value(_)),
     event((i,1,N+1),reset).
learnt_event(((i,1),1),value(true)):-
     event(((i,1),1),value(true)).
learnt_event(((i,1),N+1),value(true)):-
     event(((i,1),N),set),
     event(((i,1),N+1),value(true)).
learnt_event(((i,1),N),value(false)):-
     event(((i,1),N),reset),
     event(((i,1),N+1),value(false)).
learnt_event(((i,1),N+1,value(Value)):-
     event(((i,1),N),value(Value))),
     event(((i,1),N+1),value(Value)).
```

A suggested way of forming the initial hypothesis is to construct one horn clause for each different grounding of the event message and to introduce a variable for the event identifier in all the constructed clauses. The body of each clause includes only the event itself with a variable identical to the one ion the head as the event identifier. This corresponds to believing that an event can be executed at any time without any constriction on the event identifier and is the most general practical hypothesis for any event. Any induced program will be a specialisation of this hypothesis, and this somewhat restricts the problem of learning a correct theory.

An example initial hypothesis for the example above is given below.

- Σ_0 =

```
learnt_event(Id,set):-
     event(Id,set).
learnt_event(Id,reset):-
     event(Id,reset).
learnt_event(Id,value(true)):-
     event(Id,value(true)).
learnt_event(Id,value(false)):-
     event(Id,value(false)).
```

A method for finding a suitable initial hypothesis, Σ_0 and an ILP algorithm that can refine Σ_0 to Σ are currently under construction. Since the task is to generalise from only positive examples, an approach that weights generality and complexity of hypotheses such as that taken in Progol [10] is considered.

An interesting additional approach is to test new hypothesis by experimenting, as in the work done with the CAP system [5]. This technique is only considered as an additional method since an agent may not have the resources to experiment if it is trying to optimise its functionality. It is also necessary to reson about failed events to use this method.

A third approach is to use results from the area of reinforcement learning. A combination of declarative programming and reinforcement learning has been suggested [2].

8.3 Executing Synthesised *Eel* programs

To be able to execute directly the synthesised programs, ground event identifiers must be substituted into the programs for any existing event identifiers that are described by variables. By substituting only the identifier of the last event to be executed in the synthesised program with the identifier of a new, not previously executed event, a query is constructed that will only succeed if all the other events can be satisfied by previously executed events.

If an increasing number of new event identifiers is substituted for the event identifiers described by variables in the synthesised program, it is possible to effectively take advantage of previous events and reduce the number of new events needed to a minimum.

Acknowledgements

This work has been done as a part of the author's research for a PhD. It has been made possible partly by a loan and a grant from the State Educational Loan Fund, Norway and partly through a scholarship from the Department of Computer Science at Bristol University. The author wishes to thank Christophe Giraud-Carrier and Steve Gregory for insightful comments.

References

[1] J. A. Davila. *Agents in Logic Programming*. PhD thesis, Department of Computing, Imperial College, London University, 1997.

[2] S. Dzeroski, L. DeRaedt, and H. Blockeel. Relational reinforcement learning. In *Proceeding of the 8th International Conference on Inductive Logic Programming (ILP-98)*, Lecture Notes in Artificial Intelligence 1446. Springer Verlag, 1998.

[3] T. H. Fung and R. A. Kowalski. The IFF Proof Procedure for Abductive Logic Programming. *The Journal og Logic Programming*, 33(2):151–166, 1997.

[4] 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.

[5] D. Hume and C. Sammut. Applying inductive logic programming in reactive environments. In S. Muggleton, editor, *Proceedings of the First Inductive Logic Programming Workshop*, 1991.

[6] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, (4):67–95, 1986.

[7] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[8] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press/MIT Press, 1995.

[9] S. Muggleton. *Inductive Logic Programming*. Number 38 in The APIC series. Academic Press, 1992.

[10] S. Muggleton. Learning from positive data. In *Proceedings of the Sixth Inductive Logic Programming Workshop*, pages 225–244, 1996.

[11] 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.

[12] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319. AAI Press, 1995.