Event Management for the Development of Multi-agent Systems using Logic Programming

Natalia L. Weinbach[†]
nlw@cs.uns.edu.ar

Alejandro J. García
ajg@cs.uns.edu.ar

Laboratorio de Investigación y Desarrollo en Inteligencia Artificial (LIDIA)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Av. Alem 1253, (8000) Bahía Blanca, Argentina
Tel: (0291) 459-5135 / Fax: (0291) 459-5136

Abstract

In this paper we present an extension of logic programming including events. We will first describe our proposal for specifying events in a logic program, and then we will present an implementation that uses an interesting communication mechanism called "signals & slots". The idea is to provide the application programmer with a mechanism for handling events. In our design we consider how to define an event, how to indicate the occurrence of an event, and how to associate an event with a service predicate or handler.

Keywords: Event Management. Event-Driven Programming. Multi-agent Systems. Logic Programming.

1 Introduction

An event represents the occurrence of something interesting for a process. Events are useful in those applications where pieces of code can be executed automatically when some condition is true. Therefore, event management is helpful to develop agents that react to certain occurrences produced by changes in the environment or caused by another agents. Our aim is to support event management in logic programming. An event can be emitted as a result of the execution of certain predicates, and will result in the execution of the service predicate or handler. For example, when programming a robot behaviour, an event could be associated with the discovery of an obstacle in its trajectory; when this event occurs (generated by some robot sensor), the robot might be able to react to it executing some kind of evasive algorithm. Another popular example is provided by modern user interfaces, where a number of small graphical devices (called widgets) are displayed to mediate human-computer interaction. By operating on such widgets the user generates an event.

In this paper we present an extension of logic programming including events. We will first describe our proposal for specifying events in a logic program, and then we will present an implementation that uses an interesting communication mechanism called "signals & slots". Such mechanism is included in a library, known as Qt [8, 9, 10], that is being widely used

Partially financed by SeCyT Universidad Nacional del Sur (subsidy: 24/ZN09) and Agencia Nacional de Promoción Científica y Tecnológica (PICT 2002 No. 13096)

[†]Supported by a research fellowship of Universidad Nacional del Sur

nowadays. The idea is to provide the application programmer with a mechanism for handling events. In our design we consider how to define an event, how to indicate the occurrence of an event, and how to associate an event with a service predicate or handler.

2 The Importance of the Use of Events

An *event* can be considered as something that comes, arrives, or happens. Particularly, in computer science, an event is an occurrence or happening of significance to a task or program, such as the completion of an asynchronous input/output operation. A task may wait for an event or any of a set of events or it may (request to) receive asynchronous notification (a signal or interrupt) that the event has occurred. [5]

When programming without asynchronous notification of events, the program is always in control of when the input occurs. While the program is looking for a particular kind of input, be it a mouse click or a key press, it isn't doing any other work. That is usually called *busy-waiting*, and wastes CPU cycles.

In some cases, programs are structured conveniently as *reactive systems*, i. e., system where given events occurring in the environment cause certain program fragments to be executed [4]. Events, that can be generated in any given state, are defined by the context. They are emitted when certain conditions hold. The entire application can be viewed as a system that reacts to events by dispatching them to the appropriate piece of code that is responsible for handling the event. What we need is a program structure that allows us to respond to a multitude of possible inputs, any of which may arrive at unpredictable times and in an unpredictable sequence. Event-driven programming supports the decomposition of the problem into a set of relatively simple event checking and service routines.

In event-driven programming, a program is structured in terms of events. Methods are associated with certain events, and they are invoked when these events occur. Therefore, the order of execution is not known in advance. Under the event-driven programming model, the program structure is divided into two rough groups, *events* and *services*. An event represents occurrences, or changes of states, or requests that the program needs to handle. A service is what you do in response to the event. While events most often originate from the outside your program, such as a mouse click, events can also be generated by other parts of the program. The program executes by constantly checking for possible events and, when an event is detected, executing the associated service.

Services should be very compact functions that initiate the required action and quickly return. This allows the program to get back to checking for other events. The core assumption in event-driven programming is that you can check for events quickly enough so that none are missed. This can only happen if both the event checkers and the service routines execute quickly. Neither event checkers nor service routines should enter into an indefinite loop.

As a consequence of asynchronism, the programmer can define what to do in the presence of an event, but he can't control when to do it. That is, the moment of execution of the code responsible for handling the event is neither fixed nor immediate.

From the point of view of simulation of discrete-events, an event is considered as an instantaneous occurrence that may change the state of the system [1]. The term *endogenous* is used to describe activities and events occurring within a system, and the term *exogenous* is

used to describe activities and events in the environment that affect the system. In general, this kind of systems have *real-time* constraints. If this timeliness is not taken into account, i. e., the treatment of the event is not immediate, the conditions that generated the event may disappear.

3 Events in Logic Programming

Our aim is to support event management in logic programming. Events are useful in those applications where code fragments can be executed automatically when some conditions become true. As far as we know, there is no Prolog implementation supporting events. However, what has been developed on logic programming languages is the handling of *exceptions* (see section 7). Since, the terms event and exception can be misused or confused, the definition we adopt for exception is that of an abnormal event which may cause the failure of the program.

To add event management support to a logic programming language we propose the definition of two layers: *specification* and *implementation*. The top layer is the one that is viewed by the user. It deals with the specification of events, and provides the basic functions to define an event, emit an event, and associate a handler to an event. The other layer is the one that implements event management, and concerns detection and handling. The implementation layer defines how the application detects events, how the control is passed to the handler routine, and how the execution continues. We are going to return to the discussion of these layers later.

The semantics of exceptions is usually well defined in the literature. However, it is not the case for event management. Taking into account that exceptions are abnormal conditions, the execution of the program code will be interrupted. Then, the control is passed to the routine that handles the exception. This routine can change the conditions of the procedure that raised the exception, so if it's re-executed it will complete normally. Therefore, we have several options to return the control to the application. The first one is not returning, that is, to abort the execution. In other case, we can execute again the instruction that raised the exception, re-execute from the beginning the procedure that raised the exception, or continue the execution with the sentence that follows the one that raised the exception.

Although we have not found a clear semantics for event management, it is easy to see that there are at least two choices:

- Interrupt the procedure that emits the event and switch to execute the service code. When the service ends its execution, the control is returned to the original procedure, like in a procedure call.
- Continue with the execution of the procedure that emits the event and execute the service *in parallel*. That would be possible if several execution units are available; otherwise, the result would be a concurrent execution.

3.1 Specification

An event can be emitted as a result of the execution of certain predicates, and will result in the execution of the service predicate. In that way we are suggesting an approach that is a hybrid programming solution, which is between traditional logic programming and event-driven programming. Our idea is to provide the programmer with tools for:

- define an event
- emit an event
- connect a service to a defined event
- disconnect a service from an event

In our approach the following predicates will cover the items mentioned above:

`ev_define(Event_Name, Arity)`

Defines an event `Event_Name/Arity`. It's mandatory to define the event before using it as an argument of `ev_emit/1`, `ev_connect/4` or `ev_disconnect/4`. The event can be a constant (`Arity` = 0) or a Prolog term (`Arity` > 0). This predicate must be inserted in the program code as a fact.

`ev_emit(Event)`

Emits the event `Event` which has been previously defined (`Event` has the form `Event_Name/Arity`). This action will execute the handlers connected to this event.

`ev_connect(Event_Name, ArityE, Handler_Name, ArityH)`

Associates an `Event` to the `Handler` predicate. The arguments `Event` and `Handler` must agree in the number and type of their parameters in case they are a Prolog term, or they must be the same constant (`ArityE` = `ArityH`).

`ev_disconnect(Event_Name, ArityE, Handler_Name, ArityH)`

Deletes a previous association between `Event` and `Handler`.

3.2 Implementation

At the implementation stage two separate tasks will be considered: Detection and Management.

Detection involves the identification of the event, i. e., the recognition that the event has happened. This usually implies a loop, continually checking for events, and probably associated to an event queue. As events happen, they are placed into an event queue to wait. Events should be removed from the queue as soon as possible. Moreover, some priority scheme can be applied to the queue, to distinguish the most critical events.

Management deals with the control switch to the handling predicate. The system has to find the correct procedure and then it must pass the control of execution, with the appropriate information about the event.

For the implementation of these tasks there are several options:

1. Check for the event in the program code, where it's sure it could occur.
2. Hold a separate process with an event loop in background that will pass the control to the appropriate handler when an event occurs.
3. Use the same detection mechanism as in the previous point, but now threads are used to execute the handler.

The first option is not real event management. It could lead to a great amount of time waiting and therefore doing no useful work (i. e., wasting CPU cycles). It adds the restriction that our events must be predictable, which is not always posible. Figure 1 illustrates what happens in this case: the main program enters in a waiting loop, looking for the arrival of event1. When event1 occurs, the control is passed to the handler. The call to and return from the handler are drawn with a dashed arrow.

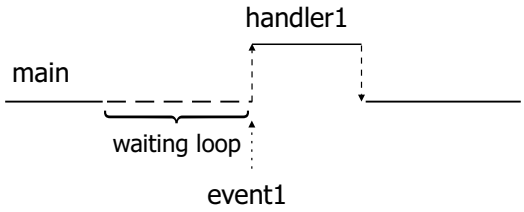

Figure 1: Event implementation with busy waiting.

The second option is the classical solution in event-driven models. To manage events, they install an event loop (a base function which embeds, controls and serializes anything else). While the loop is running, two tasks have to be performed again and again: events have to be recognized and associated functions have to be called to handle them. For example, in object oriented programming this is implemented using *Listeners*. They are objects that correspond to events, and include methods that are called in response to an event being generated. The event loop checks the queue and if there are events waiting, it will take the first waiting event off the queue, look up the listener(s) for this event, and call the method that each listener uses to handle this particular type of event. When a new event raises inside the handler there are two approaches: the event is handled once the current handler completes execution, or the new handler associated is executed immediately like in a cascaded execution (when it finishes, the previous handler will resume). Figure 2 shows an example with the second approach.

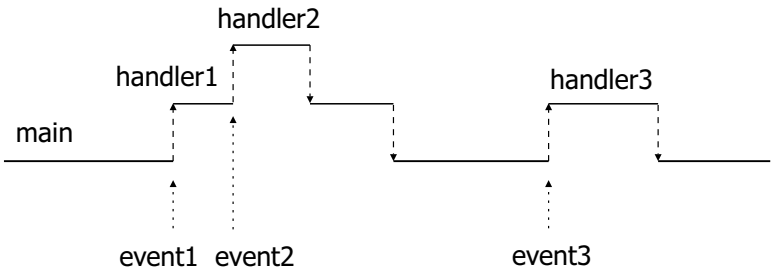

Figure 2: Event implementation with a queue.

In the third option, the intention is to have a main event loop running on the principal thread. If an event occurs, a new thread will be created for each associated handler. This implementation allows a concurrent execution on a computer with one processor, or a parallel execution, provided that we have more than one processing element (see Fig. 3).

In this work we undertake the implementation of the second option. This implementation can be done in two ways. One approach is coding it on the underlying Prolog implementation, and therefore providing event management through Prolog primitives. The other approach is to use a foreign library mechanism and link it with a Prolog file. In this paper we adopt this last approach making use of the *signals and slots* mechanism of the Qt toolkit.

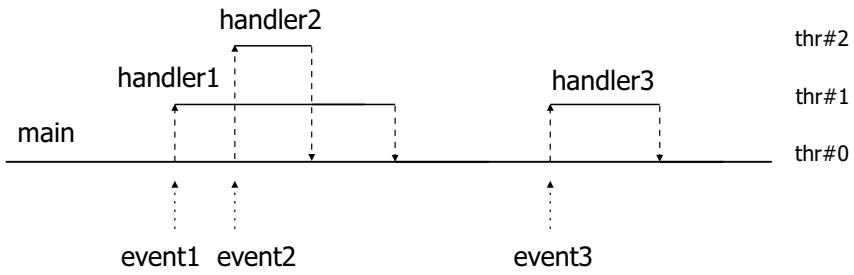

Figure 3: Event implementation with threads.

4 Qt toolkit

Qt is a multi-platform C++ GUI toolkit created and maintained by Trolltech [8]. Unlike MFC (Microsoft Foundation Classes) and similar toolkits, it runs on all 32-bit Windows platforms (apart from NT 3.5x), most Unix variants, including Linux, Solaris, AIX and Mac OS X, and embedded platforms. It provides application developers with all the functionality needed to build applications with state-of-the-art graphical user interfaces. Qt is fully object-oriented, easily extensible, and allows true component programming. [10]

4.1 Signals & Slots

Qt introduces an innovative alternative for inter-object communication, called "signals and slots", that replaces the old and unsafe callback technique [9]. The signal/slot mechanism resemble the Java listener approach, except that it is more lightweight and versatile. A signal is emitted when a particular event occurs. A slot is the handler of the event, i. e., a function that is called in response to a particular signal. Qt's widgets have many pre-defined signals and slots, but it is common practice to add your own slots so that you can handle the signals that you are interested in.

All classes that inherit from `QObject` or one of its subclasses can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to the outside world. This is all the object does to communicate. It does not know or care whether anything is receiving the signals it emits. This is true information encapsulation, and ensures that the object can be used as a software component.

The signals and slots mechanism is type safe: the signature of a signal must match the signature of the receiving slot. You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you desire. It is even possible to connect a signal directly to another signal. Since slots are normal member functions with just a little extra spice, they have access rights like ordinary member functions. A slot's access right (private, protected, public) determines who can connect to it.

In general, signals are *synchronous*. When a signal is emitted, the slots connected to it are executed *immediately*, just like a normal function call. The signal/slot mechanism is totally independent of any GUI event loop. You emit a signal by using `emit signal(<arguments>)`. The `emit` will return when all slots have returned. If several slots are connected to one signal, the slots will be executed one after the other, in an arbitrary order, when the signal is emitted.

The signals and slots mechanism can be used in separate threads, as long as the rules for `QObject` based classes are followed. The slots are executed in the thread context that emitted the signal. But, the programmer has to explicitly indicate where the new thread is created, where it resumes, and also he must ensure a correct synchronization.

Together, signals and slots make up a powerful component programming mechanism. The signals and slots mechanism is efficient, but not quite as fast as "real" callbacks. Signals and slots are slightly slower because of the increased flexibility they provide, although the difference for real applications is insignificant. In general, emitting a signal that is connected to some slots, is approximately ten times slower than calling the receivers directly, with non-virtual function calls. This is the overhead required to locate the connection object, to safely iterate over all connections (i. e., checking that subsequent receivers have not been destroyed during the emission) and to marshall any parameters in a generic fashion. The simplicity and flexibility of the signals and slots mechanism is well worth the overhead, which users won't even notice.

5 Implementing Event Management in Logic Programming

As stated in the previous sections, events are useful in those applications where pieces of code can be executed automatically when some condition is true. Examples of these applications include reactive systems (e. g., temperature controller) and software agents in general, working cooperatively or competitively. For instance, when programming robots it can be desirable that they react moving back or spinning as soon as an obstacle is found. Also, events can be raised as part of communication actions.

In our approach we will make use of the Qt toolkit to include event management in Prolog (this idea was already explored in [11]). Since Qt is written in C++, we have to choose a Prolog implementation that provides a bidirectional language interface between C and Prolog, so that C functions can be called from Prolog and Prolog predicates can be invoked from C. Particularly, we use SWI-Prolog [13] for our implementation. The base mechanism is illustrated in Fig. 4.

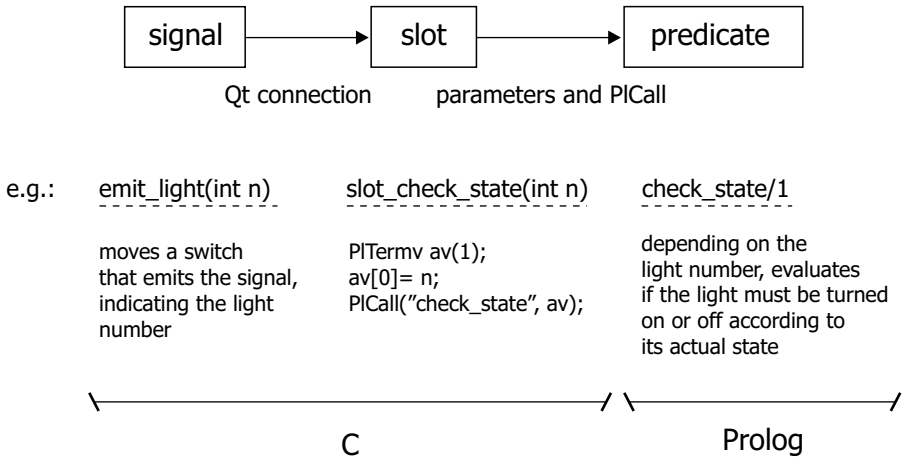

Figure 4: Signals and Slots in Prolog.

An application programmer should write his programs that use events without concerning about the language interface or the operation of Qt's signals and slots. Therefore, we have

implemented a pre-processor that uses a Prolog program as an input and works on behalf of the programmer to generate some output files. These automatically generated files are then compiled with `gcc` to produce a shared object (a library) that will be dynamically linked to the Prolog program to implement event management.

5.1 Language Interface and Pre-processor Details

To use the Qt toolkit for event management, we'll need to parse the Prolog file and insert the proper declarations and functions in a C++ file. This is the task of our *pre-processor* program.

Given a Prolog program that includes events, e. g. `test.pl`, the pre-processor generates three new files: `test_aux.pl` −with extra predicates−, `test.h` and `test.cpp` −header and implementation C++ files. The file `test.h` will contain the signal definitions and the headers of the procedures and functions included in the implementation.

All Prolog arguments will be treated in C++ as `PlTerm` variables. The class `PlTerm` is defined by the language interface to represent a Generic Prolog term. It provides constructors and operators for conversion to native C-data and type-checking [12].

When the pre-processor is faced with the fact `ev_define(Event_Name, Arity)`, it does the following tasks:

- defines the signal in `test.h` as `void Event_Name()` if the event is a Prolog constant (`Arity` = 0), or as `void Event_Name(PlTerm, PlTerm, ..., PlTerm)` if the event consists of a term with `Arity` arguments (it matchs the number of PlTerm variables).
- defines a procedure to emit the particular event, which calls Qt's `emit` with the name of the signal.
- defines a predicate that calls the procedure declared in the item above, so that a Prolog application can emit an event.
- adds a new rule in Prolog (`test_aux.pl`) that maps a call to `ev_emit` to the name of the actual event-emitter predicate (the one defined in the previous item).

In the presence of `ev_emit(Event)` the pre-processor does nothing. That is, no extra definition of functions or predicates is needed. At execution, the call to `ev_emit(Event)` will map to the actual name of the predicate that emits the event in `test_aux.pl`, and this last predicate will call the service of the C++ class that finally emits the event.

With `ev_connect(Event_Name, ArityE, Handler_Name, ArityH)` the pre-processor makes changes in the files, it:

- implements a procedure to connect the event to the handler using Qt's connect: `QObject::connect(this, SIGNAL(<Event>), this, SLOT(<Handler>))`, and defines a predicate to call this procedure.
- defines the slot writing a procedure that calls the Prolog handling predicate through a `PlCall` primitive.

- adds a mapping from `ev_connect` to the name of the predicate defined in C++ that makes the connection.

The presence of an `ev_disconnect(Event_Name, ArityE, Handler_Name, ArityH)` call produces analogous transformations using Qt's disconnect: `QObject::disconnect(this, SIGNAL(<Event>), this, SLOT(<Handler>))`, without defining the slot function.

Initialisation and error conditions deserve special consideration and will be explained in the next sections.

5.2 Initialisation

Foreign modules will be linked using *dynamic linking*. In this kind of linking, the extensions are linked to a shared library (`.so` file on most Unix systems) and loaded into the running Prolog process.

The predicate `load_foreign_library/2` searchs for the given foreign library and links it to the current SWI-Prolog instance. The library may be specified with or without the extension. Its syntax is:

```
load_foreign_library(+Lib).
```

To begin using the shared library in our implementation, the user must call `load_foreign_library(test)` and `ev_init`. The predicate `ev_init/0` makes the initialisation needed by the event management mechanism. Therefore, the last sentences of our `test.pl` file might be:

```
:- load_foreign_library(test).
:- ev_init.
```

5.3 Error conditions

An `ev_emit` will execute *all* the handlers connected to this event. In the execution of the handlers, Prolog backtracks to see if the handler can succeed. If the handler fails, there is no error notification (i. e., nothing is written on the screen). Anyway, the execution continues with the next handler.

In the call `ev_emit(Event)` if `Event` is not defined, Prolog will answer "`No`" in case there are other events defined; otherwise Prolog will report "`ERROR: Undefined procedure: ev_emit/1`".

Beware of the number of `ev_connect` calls with the same arguments you make. If you call the same `ev_connect` goal twice, the event will be connected to the handler twice. That is, in the presence of the event, the same handler will be executed two times.

If the predicate `ev_connect/4` is invoked with a nonexistent event, such as `event_x`, Prolog will answer:

```
QObject::connect: No such signal EvMgr::ev_eventx()
QObject::connect:  (sender name:   'unnamed')
```

```
QObject::connect:  (receiver name: 'unnamed')
```

Another case is when you call `ev_connect/4` with a handler not defined. Prolog will allow you to make the connection, but when the event is emitted the error will arise:

```
?- ev_connect(eventx, 1, handlerx, 1).
Yes

?- ev_emit(eventx(X)).
ERROR: Undefined procedure: handlerx/1
   Exception: (8) emit_eventx(X) ?
```

6 Example

To clarify the use of the primitives presented, we will describe a simple example. The program involved acts as the manager of five light switches. When a particular switch is moved, an event is emitted. Then, the program calls the handler of the lights, which turns them on or off depending on their current state. The Prolog code is shown in Fig. 5.

```
% Event definition:
ev_define(light, 1).

% The handler:
check_state(X):-
    retract(light_state(X, on)),
    off(X),
    assert(light_state(X, off)).
check_state(X):-
    retract(light_state(X, off)),
    on(X),
    assert(light_state(X, on)).

% Predicates for connecting and disconnecting the event to its handler:
connect_light:- ev_connect(light, 1, check_state, 1).
disconnect_light:- ev_disconnect(light, 1, check_state, 1).

% Predicate which emits the event:
toggle(X):- ev_emit(light(X)).

% Initialisation:
:- load_foreign_library(light).
:- ev_init.
```

Figure 5: File `light.pl`

As mentioned earlier, given a Prolog program that includes events, the pre-processor generates three new files. We will show next, part of the code automatically generated from `light.pl`. The file `light.h` contains:

```
signals:
    void ev_light(PlTerm);                      – signal definition
public slots:
    void emit_light(PlTerm);                    – emits the signal
    void slot_check_state(PlTerm);              – calls the Prolog handler
    void connect_light1_check_state1();         – connects the signal with the slot
    void disconnect_light1_check_state1();      – disconnects the signal and the slot
```

The C++ implementation file, `light.cpp`, includes the translations explained in section 5.1, and provides the implementation for `light.h`. Finally, the auxiliary Prolog file, `light_aux.pl`, contains the mappings needed to differentiate between the various events and handlers:

```
ev_emit(light(A1)):- emit_light(A1).
ev_connect(light, 1, check_state, 1):- connect_light1_check_state1.
ev_disconnect(light, 1, check_state, 1):- disconnect_light1_check_state1.
```

7 Related work

As stated before, as far as we know, there is no Prolog extension considering events. However, several Prolog versions implement exceptions. ISO Prolog defines an exception handling mechanism, based on the built-in control constructs `catch/3` and `throw/1`. [3, 7]

In effect when an error occurs the current goal is replaced by a goal:

```
throw(error(Error_term, Implementation_term))
```

where `Error_term` is a term that supplies information about the error, and `Implementation_term` is an implementation defined term that can be omitted.

A goal `catch(Goal, Catcher, Handler)` is true if `call(Goal)` is true, or if an exception is raised which is caught by `Catcher` and `Handler` then succeeds.

For example, SWI-Prolog defines the predicates `catch/3` and `throw/1` for ISO compliant raising and catching of exceptions [13]. In the implementation 4.0.6, most of the built-in predicates generate exceptions, however some obscure predicates merely print a message, start the debugger and fail, which was the normal behaviour before the introduction of exceptions.

CIAO Prolog implements ISO Prolog exception handling, and also adds some predicates [2]. An interesting one is `intercept(Goal, Error, Handler)` that executes `Goal`. If an exception is raised during its execution, `Error` is unified with the exception, and if the unification succeeds, `Handler` is executed and then *the execution resumes* after the predicate which produced the exception. Note the difference with built-in `catch/3`, where the entire execution derived from `Goal` is *aborted*.

Quintus Prolog supports exceptions with the predicates `raise_exception/1` and `on_exception/3`, which are analogous to `throw/1` and `catch/3` respectively [6].

8 Conclusions and Future Work

In the present work we have presented a mechanism which allows the reaction of Prolog programs to event occurrences. The program behaviour will be defined by the events that are tested and the services executed in response to those events. For this purpose, we have used the inter-object communication method, signals & slots, of the Qt toolkit and a C-Prolog language interface.

In future work, we will deal with multi-threading support of event management, so as to evaluate the possibility of parallelism. With this model of execution, we can simulate a more realistic environment for agents, where they communicate and generate events for each other indicating the presence of certain conditions, or even messages.

References

[1] J. Banks, J. S. Carson II, B. L. Nelson, and D. M. Nicol. *Discrete-event system simulation.* Prentice-Hall, Inc., 3rd edition, 2001.

[2] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. *The CIAO Prolog System.* The CLIP Group, School of Computer Science, Technical University of Madrid, Madrid, April 2004.

[3] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard. Reference Manual.* Springer-Verlag, 1996.

[4] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* John Wiley and Sons, 3rd edition, 1998.

[5] HyperDictionary. http://www.hyperdictionary.com.

[6] Intelligent Systems Laboratory. *Quintus Prolog User's Manual.* Swedish Institute of Computer Science (SICS), Sweden, June 2003. Release 3.5.

[7] JTC1/SC22/WG17. ISO Prolog standard: ISO/IEC 13211-1, 1995. http://pauillac.inria.fr/~deransar/prolog/docs.html.

[8] Trolltech. http://www.trolltech.com.

[9] Trolltech. *Qt 3.1 Whitepaper*, 2003. ftp://ftp.trolltech.com/qt/pdf/3.1/qt-white-paper-a4-web.pdf.

[10] Trolltech. *Qt Reference Documentation*, 2003. http://doc.trolltech.com/3.3/ index.html.

[11] N. Weinbach and A. García. Una Extensión de la Programación en Lógica que incluye Eventos y Comunicación (An Extension of Logic Programming that includes Events and Communication). *VI Workshop de Investigadores en Ciencias de la Computación (WICC 2004)*, pages 379–383, May 2004. Universidad Nacional del Comahue, Neuquén, Argentina.

[12] Jan Wielemaker. *A C++ interface to SWI-Prolog.* Amsterdam, 1990-2002.

[13] Jan Wielemaker. *SWI-Prolog 5.2 Reference Manual.* Amsterdam, Nov. 2003.