Event-driven programming

The addition of event-driven programming capacities to the Logtalk system is based on a simple but powerful idea [Moura 94]:

> The computations must result, not only from message sending, but also from the **observation** of message sending.

The need to associate computations to the occurrence of events was very early recognized in several knowledge representation languages, in some programming languages [Stefik 86, Moon 86], and in the implementation of operative systems [Tanenbaum 87] and graphical user interfaces.

With the integration between object-oriented and event-driven programming, we intend to achieve the following goals:

- Minimize the coupling between objects. An object should only contain what is intrinsic to it. If an object observes another object, that means that it should depend only on the (public) protocol of the object observed, and not on the implementation of that same protocol.

- Provide a framework for building reflexive systems in Logtalk based on the dynamic behavior of objects in complement to the reflective information of the object's contents and relations.

Definitions

The words *event* and *monitor* have multiple meanings in computer science, so, to avoid misunderstandings, it is advisable that we start by defining them in the Logtalk context.

Event

In an object-oriented system, all computations start through message sending. It thus becomes quite natural to declare that the only event that can occur in this kind of system is precisely the sending of a message. An event can thus be represented by the ordered tuple `(Object, Message, Sender)`.

If we consider message processing an indivisible activity, we can interpret the sending of a message and the return of the control to the object that has sent the message as two distinct events. This distinction allows us to have a more precise control over a system dynamics. In Logtalk, these two types of events have been named `before` and `after`, respectively for message sending and returning. Therefore, we end up by representing an event by the ordered tuple `(Event, Object, Message, Sender)`.

The implementation of the event notion in Logtalk enjoys the following properties:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Independence between the two types of events

We can choose to watch only one event type or to process each one of the events associated to a message sending in an independent way.

All events are automatically generated by the message sending mechanism

The task of generating events is accomplished, in a transparent way, by the message sending mechanism. The user just defines which are the events in which he is interested in.

The events watched at any moment can be dynamically changed during program execution

The notion of event allows the user not only to have the possibility of observing, but also of controlling and modifying an application behavior, namely by dynamically changing the observed events during program execution. It is our goal to provide the user with the possibility of modeling the largest possible number of situations.

Monitor

Complementary to the notion of event is the notion of monitor. A monitor is an object that is automatically notified by the message sending mechanisms whenever certain events occur. A monitor should naturally define the actions to be carried out whenever a monitored event occurs.

The implementation of the monitor notion in Logtalk enjoys the following properties:

Any object can act as a monitor

The monitor status is a role that any object can perform during its existence. The minimum protocol necessary is declared in protocol `event_handlersp`. An extended protocol is available in protocol `monitorp`.

Unlimited number of monitors for each event

Several monitors can observe the same event because of distinct reasons. Therefore, the number of monitors per event is bounded only by the available computing resources.

The monitor status of an object can be dynamically changed in runtime

This property does not imply that an object must be dynamic to act as a monitor (the monitor status of an object is not stored in the object).

The execution of actions, defined in a monitor, associated to each event, never affects the term that denotes the message involved

In other words, if the message contains uninstantiated variables, these are not affected by the acting of monitors associated to the event.

Event generation

For each message that is sent (using the `::/2` message sending mechanism) the runtime system automatically generates two events. The first — `before event` — is generated when the message is sent. The second — `after event` — is generated after the message has successfully been executed.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Communicating events to monitors

Whenever a spied event occurs, the message sending mechanisms call the corresponding event handlers directly for all registered monitors. These calls are made bypassing the message sending primitives in order to avoid potential endless loops. The event handlers consist in user definitions for pre-declared public predicates (one for each event kind; see below for more details).

Performance concerns

Ideally, the existence of monitored messages should not affect the processing of the remaining messages. On the other hand, for each message that has been sent, the system must verify if its respective event is monitored. Whenever possible, this verification should be performed in constant time and independently from the number of monitored events. The events representation takes advantage of the first argument indexing performed by most Prolog compilers, which ensure — in the general case — an access in constant time.

Event-support can be turned off on a per-object (or per-category) basis using the compiler flag `events/1`. With event-support turned off, Logtalk uses optimized code for processing message sending calls that skips the checking of monitored events, resulting in a small but measurable performance improvement.

Monitor semantics

The established semantics for monitors actions consists on considering its success as a necessary condition so that a message can succeed:

- All actions associated to events of type `before` must succeed, so that the message processing can start.

- All actions associated to events of type `after` also have to succeed so that the message itself succeeds. The failure of any action associated to an event of type `after` forces backtracking over the message execution (the failure of a monitor never causes backtracking over the preceding monitor actions).

Note that this is the most general choice. If we wish a transparent presence of monitors in a message processing, we just have to define the monitor actions in such a way that they never fail (which is very simple to accomplish).

Activation order of monitors

Ideally, whenever there are several monitors defined for the same event, the calling order should not interfere with the result. However, this is not always possible. In the case of an event of type `before`, the failure of a monitor prevents a message from being sent and prevents the execution of the remaining monitors. In case of an event of type `after`, a monitor failure will force backtracking over message execution. Different orders of monitor activation can therefore lead to different results if the monitor actions imply object modifications unrecoverable in case of backtracking. Therefore, the order for monitor activation must be always taken as arbitrary. In effect, to suppose or to try to impose a specific sequence implies a global knowledge of an application dynamics, which is not always possible. Furthermore, that knowledge can reveal itself as incorrect if there is any changing in the execution conditions. Note that, given the independence between monitors, it does not make sense that a failure forces backtracking over the actions previously executed.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Event handling

Logtalk provides three built-in predicates for event handling. These predicates enable you to find what events are defined, to define new events and to abolish events when they are no longer needed. If you plan to use events extensively in your application, then you should probably define a set of objects that use the built-in predicates described below to implement more sophisticated and high-level behavior.

Finding defined events

The events that are currently defined can be retrieved using the Logtalk built-in predicate `current_event/5`:

```
| ?- current_event(Event, Object, Message, Sender, Monitor).
```

Note that this predicate will return a **set** of matching events if some of the returned arguments are free variables or contain free variables.

Defining new events

New events can be defined using the Logtalk built-in predicate `define_events/5`:

```
| ?- define_events(Event, Object, Message, Sender, Monitor).
```

Note that if any of the arguments is a free variable or contains free variables, this call will define the **set** of matching events.

Abolishing defined events

Events that are no longer needed may be abolished using the `abolish_events/5` built-in predicate:

```
| ?- abolish_events(Event, Object, Message, Sender, Monitor).
```

If called with free variables, this goal will remove all matching events.

Defining event handlers

There are two pre-declared public predicates, `before/3` and `after/3`, that are automatically called to handle `before` and `after` events. Any object that plays the role of monitor should define one or both of these event handler methods:

```
before(Object, Message, Sender) :-
    ... .

after(Object, Message, Sender) :-
    ... .
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The arguments in both methods are instantiated by the message sending mechanisms when a spied event occurs. For example, assume that we want to define a monitor called `tracer` that will track any message sent to an object by printing a describing text to the standard output. Its definition could be something like:

```
:- object(tracer).

    before(Object, Message, Sender) :-
        write('call: '), writeq(Object), write(' <-- '), writeq(Message),
        write(' from '), writeq(Sender), nl.

    after(Object, Message, Sender) :-
        write('exit: '), writeq(Object), write(' <-- '), writeq(Message),
        write(' from '), writeq(Sender), nl.

:- end_object.
```

Assume that we also have the following object:

```
:- object(any).

    :- public(bar/1) .
    :- public(foo/1) .

    bar(bar).

    foo(foo).

:- end_object.
```

After compiling and loading both objects, we can start tracing every message sent to any object by calling the `define_events/5` built-in predicate:

```
| ?- define_events(_, _, _, _, tracer).

yes
```

From now on, every message sent to any object will be traced to the standard output stream:

```
| ?- any::bar(X).

call: any <-- bar(X) from user
exit: any <-- bar(bar) from user
X = bar

yes
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

To stop tracing, we can use the `abolish_events/5` built-in predicate:

```
| ?- abolish_events(_, _, _, _, tracer).

yes
```

Copyright © Paulo Moura — Logtalk.org
XHTML + CSS Last updated on: November 16, 2005

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at **http://www.renderx.com/**