Profiling programs

In this example, we will illustrate the use of:

- events
- monitors

by defining a simple profiler that prints the starting and ending time for processing a message sent to an object.

Messages as events

In a pure object-oriented system, all computations start by sending messages to objects. We can thus define an *event* as the sending of a message to an object. An event can then be specified by the tuple `(Object, Message, Sender)`. This definition can be refined by interpreting the sending of a message and the return of the control to the object that has sent the message as two distinct events. We call these events respectively `before` and `after`. Therefore, we end up by representing an event by the tuple `(Event, Object, Message, Sender)`. For instance, if we send the message:

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

X = 1
yes
```

the two corresponding events will be:

```
(before, foo, bar(X), user)
(after, foo, bar(1), user)
```

Note that the second event is only generated if the message succeeds. If the message as a goal have multiple solutions, then one `after` event will be generated for each solution.

Events are automatically generated by the message sending mechanisms for each public message sent using the `::/2` operator.

Profilers as monitors

A monitor is an object that reacts whenever a spied event occurs. The monitor actions are defined by two event handlers: `before/3` for `before` events and `after/3` for `after` events. These predicates are automatically called by the message sending mechanisms when an event registered for the monitor occurs.

In our example, we need a way to get the current time before and after we process a message. We will assume that we have a `time` object implementing a `cpu_time/1` predicate that returns the current CPU time for the Prolog session:

```
:- object(time).

    :- public(cpu_time/1).
    :- mode(cpu_time(-number), one).

    ...

:- end_object.
```

Our profiler will be named `stop_watch`. It must define event handlers for the `before` and `after` events that will print the event description (object, message, and sender) and the current time:

```
:- object(stop_watch).

    :- uses(time).

    before(Object, Message, Sender) :-
        write(Object), write(' <-- '), writeq(Message),
        write(' from '), write(Sender), nl, write('STARTING at '),
        time::cpu_time(Seconds), write(Seconds), write(' seconds'), nl.

    after(Object, Message, Sender) :-
        write(Object), write(' <-- '), writeq(Message),
        write(' from '), write(Sender), nl, write('ENDING at '),
        time::cpu_time(Seconds), write(Seconds), write(' seconds'), nl.

:- end_object.
```

After compiling and loading the `stop_watch` object (and the objects that we want to profile), we can use the `define_events/5` built-in predicate to set up our profiler. For example, to profile all messages that are sent to the object `foo`, we need to call the goal:

```
| ?- define_events(_, foo, _, _, stop_watch).

yes
```

This call will register `stop_watch` as a monitor to all messages sent to object `foo`, for both `before` and `after` events. Note that we say "as a monitor", not "the monitor": we can have any number of monitors over the same events.

From now on, every time we sent a message to foo, the stop_watch monitor will print the starting and ending times for the message execution. For instance:

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

foo <-- bar(X) from user
STARTING at 12.87415 seconds
foo <-- bar(1) from user
ENDING at 12.87419 seconds

X = 1
yes
```

To stop profiling the messages sent to foo we use the abolish_events/5 built-in predicate:

```
| ?- abolish_events(_, foo, _, _, stop_watch).

yes
```

This call will abolish all events defined over the object foo assigned to the stop_watch monitor.

Summary

- An event is defined as the sending of a (public) message to an object.
- There are two kinds of events: before events, generated before a message is processed, and after events, generated after the message processing completed successfully.
- Any object can be declared as a monitor to any event.
- A monitor defines event handlers, the predicates before/3 and after/3, that are automatically called by the runtime engine when a spied event occurs.
- Three built-in predicates, define_events/5, current_event/5, and abolish_events/5, enables us define, query, and abolish both events and monitors.

XHTML + CSS Last updated on: November 16, 2005