Workshop: Reflexion, OOPSLA-91

OOPSLA '91 Workshop on Reflection and. Meta-Level Architectures in Object-Oriented Programming.

Event Models or Meta-Level Architectures beyond "slot-value-using-class"

Ralf Möller

Dept. of Computer Science, University of Hamburg,
Bodenstedtstr. 16, 2000 Hamburg 50, Germany,
Email: moeller@informatik.uni-hamburg.de

1 Introduction

Currently, the meta-level facilities of CLOS are often used to trace certain slot-access operations in one way or another. We call these access operations *events*. An event specifies an activity (of an object system) that causes or triggers a meta-level shift. In CLOS, events are reified by the application special generic functions (e.g. `slot-value-using-class` and friends) which dispatch an appropriate method specialized for the metaclass of the instance whose slot is to be modified. This class-specific meta-level access can be extended to an instance-specific mechanism with a few lines of code (see [1] for details). But, are these meta-level hooks enough?

In this paper we try to show that it could be useful to provide a way to combine simple events to higher-level compound events. We are interested in defining these compound events declaratively using so-called event models. Event models are used to "recognize" concepts not evident in the events reified by basic meta-level hooks. A generalization of these ideas would be to supply declarative event models for events (or activities) in the object system itself. In this case a reflective architecture might be supported. However, to explain the main points we apply meta-level facilities to the domain-level only.

The concepts described in this paper have been developed to support the generation of visualizations for (existing) object systems. The goal is to change neither the modeling nor the code of the object system. Application and visualization should be kept separate.

For presentation purposes we choose a simple constraint net application. A visualization of the net can be found in [1]. The example application has been borrowed from [5], so there is no need to go into details here. The constraint net uses CLOS classes to model stock exchange relations. Certainty intervals describing brokers' (and other participants') estimations about when a split of stocks is to be expected are propagated through a constraint net. The participants are modeled as so-called assertions. The `assertion` class defines two slots to represent certainty intervals, one for the lower bound and one for the upper bound. The dependencies between assertions are modeled by several `constraint` classes. In this paper we focus on the assertions. During propagation of estimation intervals the slots `lower-bound` and `upper-bound` of assertion objects (e.g. `broker1` and `broker2`) are modified. For our visualization these modifications are interesting events. A simple event is the access to one slot of one object.

In the next sections we describe how declarative event models can be used to provide descriptions of the events we are interested in. After a sketch of the models we develop

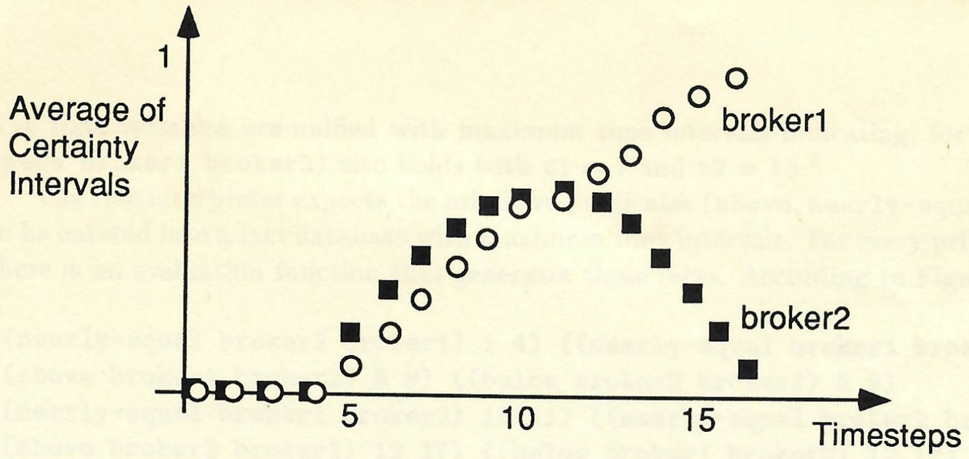

Figure 1: An example of the behavior of broker1 and broker2.

extensions to the Metaobject Protocol that facilitate the application of event models. Due to space limitations we do not consider how visualizations are actually generated.

2 Event Models

Declarative event models have been used extensively in high-level vision (see [3]) to interpret image sequences. In this paper we consider an early approach using relational models. The original work has been done with traffic scenes in mind. Nevertheless, the models can be used in more abstract domains, too.

For instance, an event might be signaled when the average of the certainty interval of an assertion object broker1 is nearly equal to that of broker2 but only if it lesser before and greater afterwards. In other words: broker1 passes broker2. Figure 1 shows an example.

A visualization might show this (temporarily) by highlighting graphic objects for the corresponding certainty intervals (e.g. using a red frame). An example of an event model for this passing event looks as follows.

```
(pass-event
  ((pass #?obj1 #?obj2) #*t1 #*t2)
  ((above #?obj1 #?obj2) #*t1 #*t3)
  ((nearly-equal #?obj1 #?obj2) #*t3 #*t4)
  ((below #?obj1 #?obj2) #*t4 #*t2))
```

The rule can be read as a Horn clause: the first term is the rule head, the others constitute the rule body. The prefixes `#?` and `#*` indicate normal variables and time-variables, respectively. Time variables are treated in a special way (s.b.). What is the (intuitive) meaning of the rules? In the rule `pass-event` an object `obj1` "passes" another object `obj2` during the interval `[t1, t2]` if the following conditions are met: from `t1` to `t3` `obj1` is "above" `obj2`, from `t3` to `t4` the objects are "nearly-equal", and from `t4` to `t2` `obj1` is "below" `obj2`. The order of the time variables is important. The rule defines implicitly: $t1 \leq t3 \leq t4 \leq t2$. `Above`, `nearly-equal`, and `below` are primitive relations, `pass` is called a derived relation. Derived relations can also be used in other rules as well. Thus, event models are hierarchical.

The rules can be used with either a forward or a backward chaining inference mechanism to derive that or to prove whether `((pass broker1 broker2) [5 9] [12 17])` holds.[1]

[1]Obviously, in an object-oriented system, it should be advantageous to create an instance of a certain event class.

The time variables are unified with maximum time intervals indicating, for instance, that `(pass broker1 broker2)` also holds with `t1` = 7 and `t2` = 15.[2]

The rule interpreter expects the primitive predicates (`above`, `nearly-equal`, and `below`) to be entered into a fact database with maximum time intervals. For every primitive relation there is an evaluation function that generates these facts. According to Figure 1 we have:

```
((nearly-equal broker2 broker1) 1 4) ((nearly-equal broker1 broker2) 1 4)
((above broker1 broker2) 5 9) ((below broker2 broker1) 5 9)
((nearly-equal broker1 broker2) 10 11) ((nearly-equal broker2 broker1) 10 11)
((above broker2 broker1) 12 17) ((below broker1 broker2) 12 17)
```

Time variables require a special treatment during the matching or unification process. Bindings of time variables are not stored in binding lists (as is usual for unified variables), but are entered into so-called time nets. These time nets encode the relations between time variables (s.a.) and allow for a consistency check of the time intervals. If an inconsistency is found during a certain step of a proof backtracking occurs. We refer to [4] for a complete discussion and focus now on the definition of time steps.[3]

The interested reader might have already noticed it. In Figure 1 we combined some slot-access events to define steps without any word of comment. The next sections discuss how the Metaobject Protocol can be used and extended to support the definition of compound events that can be interpreted as steps.

3 The Problem

Using the Metaobject Protocol, it is not difficult to have the change of slot values recorded. But, a single access (to either the `upper-bound` or `lower-bound` slot) does not really constitute an interesting event. At a first glance the event should be signaled when both slots are changed. Where is the problem?

- We know nothing about the order in which slots are modified.
- The application might decide not to call `(setf slot-value)` when the new and the old values are identical. In this case, there will be no possibility to signal an access event directly.
- Even worse, the slot `lower-bound` might be changed again before the corresponding access to its counterpart `upper-bound` occurs.

The problem is that the notion of a (propagation) step is not modeled in the program domain. If we do not want to change the domain modeling we have to define an interpretation-dependent step notion using meta-level facilities.

4 Some Ideas for a Solution

It should be clear now that we need a more sophisticated meta-level architecture (beyond `slot-value-using-class`) to link visualizations to existing applications. The next sections describe ideas that might shed some light on how to design this architecture.

[2] The term `((pass broker1 broker2) [5 9] [12 17])` will be transmitted to the visualization generator.

[3] If the events are not to be recorded an event model allowing incremental recognition might be provided.

4.1 Object Steps

In our constraint example the assertion objects have more slots than only `lower-bound` and `upper-bound`. But only access operations to the latter are relevant and have to be arranged or "synchronized". All slot-access events are signaled to a special synchronizer object which considers only modifications of certain slots declared beforehand (`lower-bound` and `upper-bound` in our case).

Suppose, the slot `lower-bound` of `broker1` had been modified. The synchronizer marked the slot `lower-bound` as changed. A so-called *object step* is generated when this slot is modified again. Just before the new slot value is entered, we collect the current values of the two slots and store them into a data structure. After generating the object step all old changed-markers are reset. The new-value is entered and the slot is marked as changed again.

It will be useful to provide a generic function that filters and maps the slot values before combining them with the object into an object step. In our constraint application the two slot values for `lower-bound` and `upper-bound` might be averaged, i.e. the object step comprises only one value. An object step is signaled by calling a generic function (e.g. `signal-object-step`) with the synchronizer and the object step as arguments. The primitive behavior could be just to print the object step in any format on the standard output stream. Special synchronizers might provide a more sophisticated administration (s.b.).

Object steps synchronize slot-access events and combine them into compound events. The definition can easily be extended to take more than two slots into consideration. The introduction of object steps avoids the problems discussed above when simple slot-access events are used directly. Notice that in this definition it is not relevant whether the corresponding other slots have been modified. It should be clear that the definition above is only one way to define object steps. It is desirable to supply different classes of synchronizer metaobjects that reflect different interpretations of object steps (e.g. a synchronizer that queues the values until all slots considered are modified). Different classes may specialize `signal-object-step` methods in different ways.

4.2 System Steps

The object steps defined above model intra-object events. But how are object steps of different objects related? Therefore, we define the conception of a "system step" as well. A system step combines object steps for a set of objects. Again, the definition is dependent on a certain interpretation of an object population and might be specialized for other systems.

System steps are compiled in nearly the same way as object steps. Object steps of different objects, say `broker1` and `broker2`, are signaled to a system synchronizer object which stores the object steps. A system step is generated when an object step is signaled for an object the system synchronizer already has an entry for. Let us suppose, this were the case for `broker1`. If there is currently no object step entry for `broker2`, an object step for `broker2` will be generated by the system synchronizer (and signaled to itself) using the current values for the `lower-bound` and `upper-bound` slot. After the current entries are disposed, a new entry for the incoming object step is set up. We do not forget to say that this definition of a system step might be extended for any number of objects.

System steps model inter-object events and can be associated with a time stamp. Providing time stamps allows the detection of much more interesting meta-level events. We are now in the position to be able to apply the event model described above. The classes and the generic functions together with their arguments cannot be presented in this contribution, but the ideas should be clear. The final section discusses the advantages of using declarative event models and of defining object or system steps in different kinds of meta-level computations.

5 Conclusion

The main advantage of integrating declarative event models into meta-level programming is that they allow more advanced events to be defined. This is due to the introduction of system steps that make possible a certain (though limited) "reasoning over time". Our architecture tries to provide a mechanism that goes beyond the current procedural facilities. Event models express higher-level concepts that do not have to be considered beforehand, e.g. by the programmer of an application.

Though not discussed primarily in this paper, it should be possible to apply the event models not only to domain-level events but also to events in the object system itself (modifications of slots of (meta)class objects, computations of class precedence lists, etc.). Following this approach could perhaps extend the current notion of computational reflection, as defined e.g. by Maes [2], especially, if it is possible to compute event models by the object system itself. This might demonstrate first steps to a more cognitive notion of reflection.

Acknowledgements

I thank my colleagues R. Cunis, V. Haarslev, G. Retz-Schmidt, and C. Schröder for valuable comments on an earlier draft of this contribution. However, all shortcomings should be charged to the author.

References

[1] Haarslev, V., Möller, R., *A Framework for Visualizing Object-Oriented Systems*, in: Proceedings ECOOP/OOPSLA'90, European Conference on Object-Oriented Programming and Object Oriented Programming Systems, Languages and Applications, Oct. 21-25., 1990, Ottawa/Canada, pp. 237-244.

[2] Maes, P., *Concepts and Experiments in Computational Reflection*, in: Proceedings OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications, Oct. 4-8., 1987, Orlando/Florida, pp. 147-155.

[3] Neumann, B., Novak, H.J., *Event Models for Recognition and Natural Language Description of Events in Real-World Image Sequences*, IJCAI-83, pp. 724-726.

[4] Novak, H.J., *A Relational Matching Strategy for Temporal Event Recognition*, in: Proceedings GWAI-84, 8th German Workshop on Artificial Intelligence, Wingst/Stade, October 1984, Laubsch, J. (Ed.), Springer, 1984.

[5] Winston, P.H., Horn, B.K.P., *Lisp - 3rd Edition*, Addison-Wesley, 1989.