Events and Sensors

Enhancing the reusability of objects[1]

POSITION PAPER

Jan Vitek,
Betty Junod, Oscar Nierstrasz
Serge Renfer and Claudia Werner

Abstract

Object-oriented programming methods promote the development of software from reusable components. In practice, reuse of object-oriented software is limited by a closed-world constraint: only components that are *compatible* – that conform to an agreed-upon protocol – may be composed. We seek to facilitate software composition. To this end, we propose an approach based on *events* and *sensors* that enhances the openness of objects, and thus increases the possibilities for their reuse.

1. Motivation

Software reusability is of paramount importance to the efficient development of large software systems. Methods for software reuse promote a way of thinking about system design that merges top-down decomposition and bottom-up composition. Using these methods, software systems are built by *composing* prepackaged reusable components [1]. Of the various approaches to software reuse, object-oriented programming (OOP) is perhaps the most promising, as it provides both mechanisms for organizing and decomposing systems into encapsulated components (i.e., objects and classes), and mechanisms for incrementally modifying and composing software (i.e., inheritance, genericity and dynamic binding) [10]. The key to the success of these mechanisms is, in all cases, the *message-passing paradigm* of OOP: any two objects that accept the same messages can be addressed uniformly by their clients even if their implementations vary. We say that an object that conforms to a client's expectations is *compatible* with the client. The potential for reuse in OOP is thus both defined and *constrained* by object-client compatibility. We seek to further enhance reusability by providing more flexible mechanisms for composing objects.

What makes composition difficult – and in some cases impossible – is the necessity for components to fit together, that is, to be compatible. In effect, OOP suffers from a closed-world assumption: only objects that conform to known interfaces can be composed. We propose an extension to the client/server paradigm supported by OOP based on *events* and *sensors* that enhances openness of objects and thus improves their potential for reuse. As part of their behaviour, objects *raise* events to signal to the environment that something of interest has occurred. Sensors are agents that monitor and respond to events by initiating actions. They can play the role of "adapters" between otherwise incompatible objects.

---

1. In *Object Management*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, July 1990, pp. 345-356

First, we shall describe a minimal object model as the framework for our discussion on the events and sensors model. Next, the basic concepts of this model are presented, followed by examples of its application. Some considerations about the implementation of such a model are discussed, references to related works are given and, finally, some prospects and conclusions are drawn.

2. A Minimal Object Model

The paradigm of object-oriented programming is itself truly polymorphic: each of its incarnations has its own set of features and redefines what it means to be "object-oriented". To facilitate our discussion of events and sensors to follow, we shall first present a minimal object model that sets out the few assumptions we make about the nature of objects. We shall adopt the definition of object-orientation formulated by Wegner [11], where:

OOP = Objects + Classes + Inheritance

Furthermore, we assume the following:

- ***Message Passing:*** Messages are the only form of inter-object communication. We define a *call/reply exchange* as a protocol composed of a *call* message from a client to an object followed (eventually) by a *reply* message from the object back to the client. We represent these messages as follows (All descriptions in this paper will be given in pseudocode, to avoid introducing a language-specific syntax. Variables begin with a capital letter and values are in lower case.):

  [call, *Server, Selector, Args*] — *Client* calls *Server*

  [reply, *Reply*] — *Server* replies to *Client*

  The *Selector* is the name of the service provided by the object (i.e., the method name). *Args* and *Reply* are the contents of the call and reply messages.

- ***Class.*** A class is the abstraction of shared characteristics of a set of objects. A class is both a static specification of behaviour and a run-time entity: it specifies responses to incoming messages and has the responsibility to generate new instances of that class. Inheritance can be viewed as the delegation to another class, the super-class, of all messages for which the class has no specified response [8].

- ***Object.*** An object is an instance of a class. It has a modifiable state consisting of a set of acquaintances — references to other objects or classes — and it communicates with other objects by call/reply exchanges. Objects may or may not execute concurrently.

3. Events and Sensors

The limitation of the object model we have outlined is that objects can only communicate if they have agreed upon a particular call/reply protocol: for one object to influence another, it must be a client that knows what requests are supported by the target. We propose a mechanism for composing objects in which a "sensor" acts as the glue between an object that can raise events and other objects that are manipulated when the events are raised.

Briefly, the approach is as follows:

1. As part of its behaviour, any object may raise a number of named *events* each of which may have a set of associated values.
2. Events are caught by the *ether*, an agent that manages the handling of events.
3. When an event is raised, the ether forwards it to a set of *sensors* monitoring that event. A sensor may initiate actions or raise events involving other objects.
4. Events may be *propagated* to sensors monitoring other events.

Events

Objects can raise events at any point while responding to a client's request. When an object raises an event it sends a message to the ether of the form:

```
[raise, Source, Event, Values]
```

where Source is the name of the object (i.e. object id), Event is the name of the event and Values is the list of event parameters. The object expects no reply to the raising of an event and, in a concurrent environment, can continue executing independently.

Sensors

A sensor consists of an execution context, parameters and an action. The context is the set of objects the sensor is acquainted with. These acquaintances are defined when the sensor is created. The parameters of a sensor are matched against those of the event. Within an action, a sensor may call its acquaintances and raise events. It is the sensor that is required to conform to the client/server protocol of the targets of its action, not the object raising the event. In this way the sensor acts as the glue between otherwise incompatible objects.

As an example, we define a sensor whose action is to print events.

```
s is a sensor
    with context Printer
    and parameters Source, Event
    and action (
        [call, Source, name, []] [reply, Sname]
        [call, Printer, print, [Sname, Event]] [reply, Isok])
```

Having defined a sensor, it can be created dynamically either as part of the behaviour of an object or by a user of the system. To create a sensor, a message with the sensor's name and context has to be sent to the ether, and a reply with a reference to the new sensor is returned.

Taking the previous example, to create sensor *s* the following messages would be exchanged (assuming that sys_printer is an object that already exists):

```
[sensor, s, sys_printer]
[reply, Sensor]
```

Ether

The ether is a mediator between events and sensors. For every <Source,Event> pair, there is a *spot* in the ether that manages the handling of occurrences of that event and remembers which sensors

are interested in the event. We say that such sensors are *tuned* to that spot. When an object raises an event, it sends a message to the corresponding spot in the ether.

The tasks of the ether include creating, maintaining and deleting both sensors and spots. Before an object can raise events, a spot in the ether must be *claimed*. The following message to the ether claims a spot for <Object,EventName>:

[claim, Object, EventName]

To register a sensor's interest for events it is necessary to tune it to spots in the ether. A sensor can be tuned to one or more spots with the following message, for Sensor and the spot <Object, EventName>:

[tune, Sensor, Object, EventName]

A sensor can be tuned to any spot but if the event parameters do not correspond to the parameters declared in the sensor's definition, the sensor does not execute and an error event can be raised.

Another property of a spot is *propagation*, that is, to re-raise the event with the same value, but at a different spot. A propagated event may be further propagated to other spots, but in this case it is clear that loops must be avoided. To set the propagation of a spot, the following message is sent to the spot <FromObject, FromEvent>:

[propagate, FromObject, FromEvent, ToObject, ToEvent]

Figure 1 shows the basic concepts of the proposed model.

Events and sensors enhance the openness of objects by delaying the client-server binding. Sensors are defined at the moment when two particular objects have to be composed but the ob-

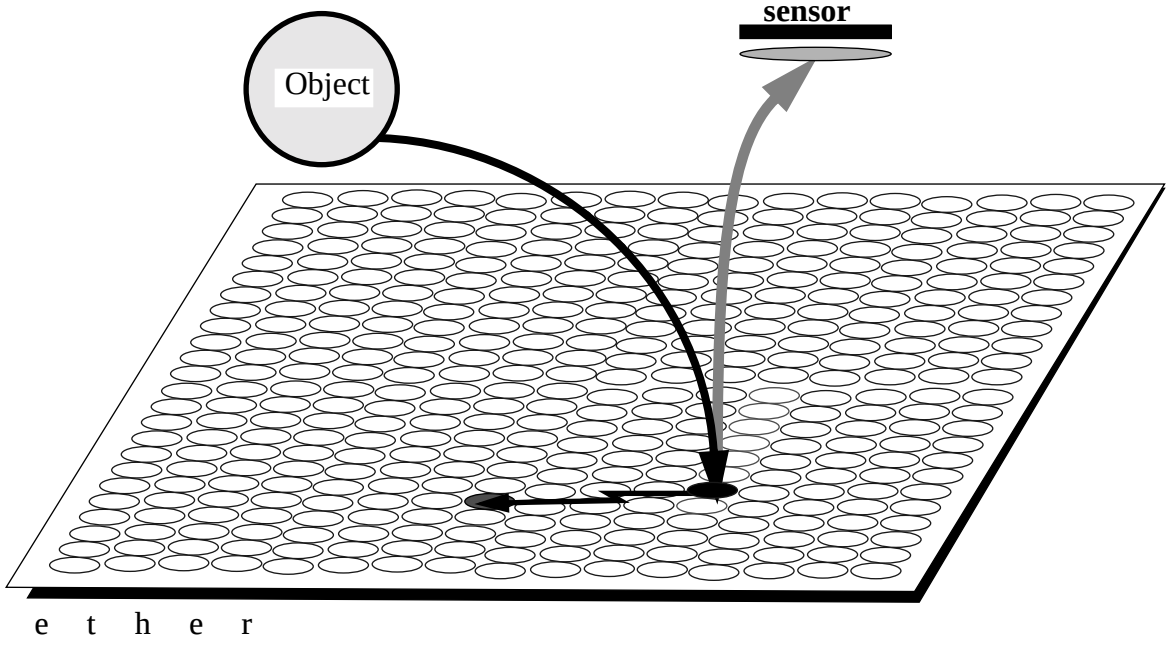

Figure 1 The ether is a collection of spots. There is a spot for each event an object can raise. When an object raises an event, it sends a message to a spot. This spot triggers sensors and propagates the event further.

jects themselves can have been built independently. It is therefore possible to put together two objects which could not otherwise be composed: a sensor catches events raised by the first object and translates them to calls that the second can understand. Sensors permit one to add behaviour specific to a particular application and to do this *a posteriori*, that is after the components have been fully specified and without modifying their implementation.

The closed-world constraint has not been entirely eliminated, since one still depends on objects to generate events to make composition possible, but a degree of freedom has nevertheless been gained.

4. Software Composition Using Sensors

We shall now demonstrate the benefits of events and sensors through examples of their use as composition mechanisms between objects. We give two complete examples and then discuss informally two other application areas. The first example shows how to connect a user-interface to an application. The second example uses propagation to solve a consistency problem when elements belonging to a collection are freed.

One application area that has always required a high degree of flexibility is user-interface construction. With libraries of graphical components, one connects standard reusable interface objects to some newly developed application objects. We assume that these two categories of objects have been developed independently. As an example let us consider an interface consisting of two buttons that are to be connected to one application object. A button is an object with a label — a string of characters — which reacts to user interaction by raising the event “clicked”. There is a button class that defines the behaviour of button instances. For each new instance, the class claims a spot in the ether for <New_button, clicked> with the message:

```
[claim, New_button, clicked]
```

When the user presses the button, the button raises an event (with no arguments):

```
[raise, Self, clicked]
```

The application object to which we wish to add a user interface expects to be called with the messages *foo* or *bar*. The user-interface to this object should therefore give the user the choice between *foo* or *bar*. Once the user has chosen one of these options, the appropriate method of the application object is called. Additionally, the choice of *foo* is to be accompanied by an audio signal. For this we define two sensors which both call the application object. The second sensor also makes a sound.

```
s1 is a sensor
        with context Appl
        and parameters Source, Event
        and action [call, Appl, bar]

s2 is a sensor
        with context Appl, Speaker
        and parameters Source, Event
        and action (
                [call, Appl, foo]
                [call, Speaker, beep])
```

The sequence of messages needed to create the buttons, the application object and the sensors is listed below. The sensors are tuned to “clicked” events coming from the buttons. It is assumed that “beeper” is a previously created object that responds to the message *beep*.

```
[call, button, create, “foo”] [reply, Foo_button]
[call, button, create, “bar”] [reply, Bar_button]
[call, Appl_obj, create] [reply, Appl]          — create an instance of App_obj
[sensor, s1, Appl] [reply, S1]                  — the context of s1 is bound to Appl
[sensor, s2, Appl, beeper] [reply, S2]          — the context of s2 is bound to Appl and beeper
[tune, S1, Bar_button, clicked]
[tune, S2, Foo_button, clicked]
```

Figure 2 shows the resulting configuration.[1] When a “clicked” event is raised by one of the buttons the corresponding sensor, S1 or S2, is activated with parameters <Foo_button, clicked, “foo”> or <Bar_button, clicked, “bar”> bound to Source, Event and Text respectively.

The second example demonstrates a use of propagation. There are cases where one desires to monitor a dynamic population of objects, that is, to listen to events originating from a *varying* number of spots. For this it is inconvenient to have to set new sensors, or tune all old ones, at each occasion a new spot is added. A simpler solution is to propagate events coming from all of these spots to a single *anchor* spot. Sensors can then be tuned to this anchor.

This scheme can be used in the implementation of a *collection* class. A collection is a container holding other objects. A collection accepts a new object when it receives the *add* message. But this new element is not encapsulated by the collection. It can still receive messages from the

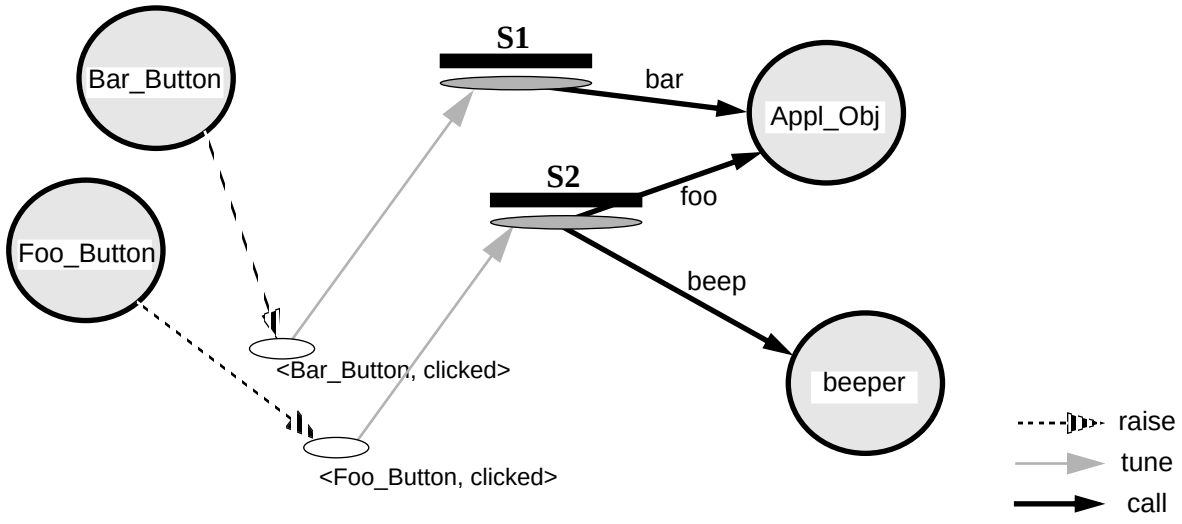

Figure 2 Two buttons are composed with an application object (App_Obj) and the “beeper” object through two sensors (S1, S2). The sensors are tuned to spots <Bar_Button, clicked> and <Foo_Button, clicked>, respectively, and react to events by sending messages.

1. The solution we present for connecting interface objects to application is, in fact, quite close to the solution adopted in the NeXTStep software kit with Interface Builder [6].

outside and can be freed without the collection being informed. This leads to dangling references.[1] To prevent this from happening, we propagate all “freed” events coming from one of the collection’s elements to a single spot known by the collection. Then, a sensor tuned to this spot removes the freed objects from the collection.

We define a sensor s with a collection object in its context and a request to this collection to remove one of its elements as the sensor’s action.

```
s is a sensor
    with context Collection
    and parameters Source, Event, Free_Object
    and action [call, Collection, remove, Free_Object]
```

When the collection class creates a new instance, part of its behaviour is to claim a spot where the “freed” events from the collections’ elements will be propagated. A sensor will be created and tuned to that spot.

```
[claim, New_Collection, element_freed]
[sensor, s, New_Collection] [reply, S1]
[tune, S1, New_Collection, element_freed]
```

Then, whenever the new collection object receives an *add* message, it sets propagation from the <New_element, freed> spot to its own <Self, element_freed> spot.

```
[propagate, New_element, freed, Self, element_freed]
```

The resulting configuration is shown in Figure 3. When a “freed” event is raised by one of the elements of the collection with the following message:

```
[raise, Self, freed, Self]
```

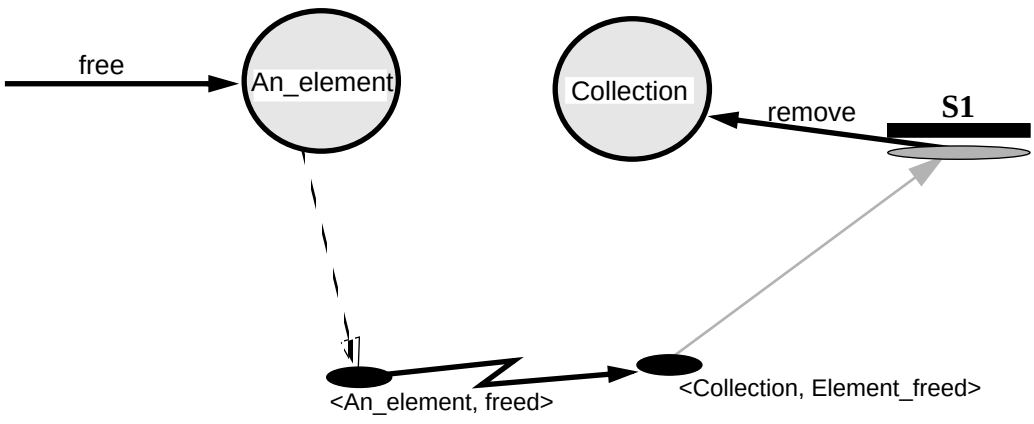

Figure 3 When an element of the collection is freed as a result of another object’s message, an event is raised and propagated to the <Collection, Element_freed> spot. S1 is triggered and its action is to remove the element from the collection.

1. Dangling references are only a problem in languages with explicit deletion (C++, Objective-C). Languages with garbage collection (Smalltalk-80, Eiffel, Lisp) are exempt from this problem.

this event is propagated to the collection's spot. This is equivalent to sending the message:

```
[raise, Collection, element_freed, Element]
```

The sensor S1 is triggered, and it sends a *remove* message to the collection with a reference to the freed object as argument.

Another application of the event/sensor model is in the implementation of a blackboard mechanism for the communication between objects. A blackboard is a place that objects can, freely, write to and read from [9]. The blackboard itself can be implemented by a unique spot, a system-wide anchor to which events can be propagated and sensors tuned. Thus objects interested in reading from the board need only to tune sensors to the anchor spot.

Events can also play a role in debugging object-oriented application. Events are natural breakpoints and sensors can be used to monitor the execution of objects. Using a simple sensor, like the one introduced in section 3 (i.e. the printer), it is possible to follow the computation of any object that raises events. Since sensors are by nature selective, it is easy to concentrate on significant aspects of the object system under investigation and filter out all the extraneous information. Sensors can also perform more complex actions, like printing the state of an object or even giving control to the user and allowing him or her to interact directly with the objects.

Although we have presented our examples in an abstract way, there are implementation considerations that are crucial to the usability of the mechanism. For example, to be really flexible, one should be allowed to define sensors on-the-fly. In the next section we will discuss some important implementation considerations.

5. Implementation Considerations

Two directions can be distinguished in the realization of software composition using sensors: the events and sensors mechanism itself and the environment supporting software composition. The first task is to design and implement the kernel for sensors. Such a kernel includes the implementation of the ether and the design of the programmatic interface used to define, create and delete sensors and spots. The second task is to provide a higher level user-interface to events and sensors. For instance, one such interface could provide a visual representation of the ether and the animation of an execution. This interface should also ease the task of defining and tuning sensors by providing a visual shorthand for some standard sensors. If the same kind of sensor is used over again, a development environment could visually represent the objects to connect by boxes and, by connecting these boxes with a particular colored line, a specific sensor would be created and tuned. We are currently developing an environment for software composition based on the paradigm of visual scripting; events and sensors are part of the features of the visual scripting tool VISTA [7]. We will now concentrate our discussion on important implementation issues of the event and sensor mechanism.

- **Efficiency**. Efficiency, or rather the lack of it, has been a major concern for many systems that use some form of triggering concept. In our case, we see optimizations taking place on two levels. First and foremost, the generation of events has to be efficient, particularly, generating unnecessary events has to be avoided. The second level deals with

finding and activating sensors. By tuning sensors to specific spots we avoid having to search among all sensors for the ones interested in a particular event; optimization can thus be focused on making their activation and execution efficient.

- **Scheduling Policy.**Our general policy towards scheduling can be stated in two points: (1) events are handled immediately after being raised, and (2) the order of execution of sensors tuned to the same spot is non-deterministic.
  However, this policy does not fit all situations, such as for example, the activation of integrity checks in databases or CAD applications. In this case, it is better to let the user decide the appropriate time for processing the integrity checks [3] [5]. This suggests the implementation of some sort of *batch mode,* where sensors are turned off and events are queued while the user is interacting with the system. At some user-defined point (e.g. at the end of a long transaction) sensors are reactivated and events are replayed from a *history file*.
  The choice of the implementation language will play an important role regarding the handling of events. If the objects can execute concurrently it is possible to have an object raise an event and continue executing. Sensors can then define concurrent threads of activity. For a sequential language[1] there are two possibilities: process the events as soon as they are raised with the object blocking until the event is processed[2], or buffer them and poll for events at certain intervals [4]. For our purpose, the polling model has two disadvantages: the event loop is not transparent to an application designer and the time between polling phases can be arbitrarily long.

- **Referential integrity**. Ensuring the integrity of a sensor's context presents some difficulties. A context has been defined as the set of acquaintances of one sensor. What would happen if one of these acquaintances became, for any reason, invalid? We have to distinguish between systems with and without garbage-collection. In systems without garbage-collection any object referenced in the context of a sensor can be freed at any time, which automatically invalidates the sensor's action. In garbage collected systems a reverse problem occurs: no object can be destroyed as long as there is a single reference to it. This means that as long as there are sensors referencing an object in their context, this object cannot be freed. We propose that the context be composed of "soft" references, that is, references to objects that do not prevent them from being freed but, whenever this occurs, dependent sensors should be deactivated.

- **Extensibility.** A major requirement for composition is to be able to define new sensors on-the-fly, while the system is executing. This implies the availability of either incremental compilation or interpretation of the sensor's action.

- **Generation of events.** What events, with what kind of arguments, should be generated? At what point in the object's computation should they be generated? Where are spots

1. We only take into account events generated by objects inside the application; external events like system interrupts are treated separately.

2. Note that recursive events on the same spot will result in an event's processing being terminated before that of its predecessor.

claimed? Who codes this? These questions will play an important part in determining the reusability of objects. Most events should be placed by the class designer since they are a integral part of class design and care should be taken not to raise events when the object is in an inconsistent state. It is conceivable to generate automatically certain standard events, for example at the beginning and the end of methods.

- **Interface specification.** The events generated by an object, and to a lesser extent the sensors this object creates, are part of the interface of the class and should therefore appear in any description or documentation of the class.

This list outlines some outstanding problems. Further investigation of related work and experimentation with prototypes are necessary before we can give satisfactory answers to these problems.

6. Related Work

Many of the concepts we have discussed are familiar to us from various other domains, such as databases, operating systems, office systems and distributed systems.

Dittrich *et al*. have developed a generalized event/trigger concept as the basic support mechanism for semantic rules in advanced database applications allowing for flexible checking times and arbitrary actions in case of rule violation [5]. There, an extension and generalization of existing trigger concepts is made so that actions are defined and associated with arbitrary events not only to some limited situations (i.e., the begin or end of a DB operation, or the occurrence of certain DB state or state transition).

A common space communication model, called *Generative Communication,* has been proposed by Gelernter [2] as a fundamental communication model for distributed systems. When two processes communicate, the data-producing process generates a new data object, called a *tuple,* and sets it adrift in a region called *tuple space*. The receiver process, having made a withdrawal request, may now access the generated tuple. Similarly, processes which create other concurrently executing processes generate ‘live tuples’ in the tuple space. Live tuples carry out some specified computation of their own and then turn into ordinary data tuples.

Implementations of KNO (KNowledge acquisition, dissemination and manipulation Objects)[9] environments enhance object-oriented systems with objects that are active, nomadic, adaptive entities. Triggers have been introduced in these systems as a feature which serves to initiate activities in objects enabling them to respond to events occurring in their environment.

All these approaches attempt, in a way, to provide a framework which satisfies the requirements of heterogeneous and evolving systems. It is in this context that we seek to facilitate software composition by providing more flexible mechanisms to enhance compatibility of components.

7. Prospects and Conclusions

Although we have proposed an approach to enhance the *openness* of objects for the purpose of software composition, we can consider the use of these concepts in other domains, as for example: exception handling mechanism and customization for a workspace.

Exception handling mechanisms are needed when situations like range errors, incorrect calls, protocol errors, etc., occur. We note, however, that these exceptions are not particular to any individual object of a class (i.e. the same range error can occur in any instance of a class). Therefore, we could think about events being propagated from instances to their class and a unique sensor being tuned to the class. In this way, all exceptions raised by any instance of the same class would be handled in the same manner. Apart from being a natural solution to the problem, this approach has advantages over other exception handling techniques by providing facilities such as: the parametrization of sensors, handling of recursive exceptions and dynamic redefinition of response.

The use of sensors can also be envisioned as a customization mechanism for a workspace. Until now most of the customization in Unix environments is done at the level of shell scripts and environment variables. If applications are objects themselves it is possible to put, for example, a sensor on the mail application. This sensor could warn the user of new mail by any audio or visual cue deemed appropriate. It could even check mail against certain user-defined patterns and start a text editor when mail meets these criteria.

The event/sensor model contributes to software composition by providing another perspective for class design and application design. We are currently investigating *teams* as a paradigm for high-level structuring of applications. Such a paradigm is an organizing principle for objects of an application. A team consists of a set of objects dedicated to a special goal according to a set of local event/action rules. A *team manager* is the representative of a group of cooperating objects. It distributes responsibilities and coordinates communication inside the team and with the outside community. While objects within a team may be accessed from outside, the events they raise are internal, i.e. visible only inside the team. The team manager may raise events as part of an action for the benefit of other teams. This paradigm is an abstraction of our model in the sense that the context of a set of sensors is embodied in the set of objects of the team, while the sensors' actions correspond to the action part of the rules.

References

[1] T. Biggerstaff and C. Richter, "Reusability Framework, Assessments, and Directions", IEEE Software, March 1987.

[2] N. Carriero and D. Gelernter, "Linda in Context", Communications of the ACM, vol. 32, no. 4, April 1989.

[3] U. Dayal, A. P. Buchmann and D. R. McCarthy, "Rules Are Objects Too: A Knowledge Model For An Active, Object-Oriented Database System", in *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1988.

[4] R. D. Hill, "Event-Response Systems - A Technique for Specifying Multi-Threaded Dialogues", CHI + GI, ACM, 1987.

[5] A. Kotz, K.R. Dittrich and J.A. Mülle, "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism", Proceedings International Conference Extending Database Technology, Springer-Verlag, Venice, 1988.

[6] NeXT 1.0 Preliminary Technical Documentation.

[7] O.M. Nierstrasz, L.Dami, V.deMey, M.Stadelmann, D.Tsichritzis and J.Vitek, “Visual Scripting - Towards Interactive Construction of Object-Oriented Applications,” in *Object Oriented Development*, ed. D.C. Tsichritzis, Centre Universitaire d’Informatique, University of Geneva, July 1990.

[8] L.A.Stein, “Delegation is Inheritance”, ACM SIGPLAN Notices, Proceedings OOPSLA 87, vol. 22, no. 12, pp. 138-146, Dec 1987.

[9] D.C. Tsichritzis, E.Fiume, S.Gibbs and O.Nierstrasz, “KNOs: knowledge acquisition, dissemination and manipulation objects,” ACM Transactions on Office Information Systems, vol. 5, no.1,pp. 96-112, Jan 1987.

[10] D.C. Tsichritzis and O.M. Nierstrasz, “Application Development Using Objects”, in *Information Technology for Organizational Systems*, Proceedings EURINFO’88, Elsevier Science Publishers B.V. 1988.

[11] P. Wegner, “Dimensions of Object-Based Language Design,” ACM SIGPLAN Notices, Proceedings OOPSLA’87, vol. 22, no. 12, pp. 168-182, Dec 1987.