Autonomous Objects: A Natural Model for Complex Applications*

*Alfons Kemper** *Peter C. Lockemann*+ *Guido Moerkotte*+ *Hans-Dirk Walter*+

*Fakultät für Mathematik und Informatik
Universität Passau
Innstrasse 33
D–94030 Passau, F. R. G.
kemper@fmi.uni-passau.de

+Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten 5
D–76128 Karlsruhe, F. R. G.
lockemann|*moer*|*walter*@ira.uka.de

Abstract

Object-oriented database systems are emerging as the "next generation" DBMSs for advanced applications, e.g., VLSI design, mechanical CAD/CAM, software engineering, etc. However, a close analysis indicates that the requirements imposed by these application domains cannot be met by an object-oriented model that relies purely on passive objects. In this work we go beyond the conventional *single-thread-of-control* paradigm of passive object models and propose a model of *active* objects which can autonomously initiate responses to environmental changes. *Autonomous* objects cooperate with each other by *synchronous* or *asynchronous* message passing—giving rise to the *multiple-thread-of-control* in such an environment. It is shown how *events*—to which active objects react—can be incorporated into this model. We propose a nondeterministic computational model for the individual active objects that allows the autonomous reaction upon events. We show that this very sparse extension to an object-oriented model gives rise to several high-level features which can be controlled by events. The object-oriented paradigm allows one to isolate the rules according to which events are being raised. This leads to a potentially rather efficient execution model compared to existing relational concepts, which are typically globally-defined event trigger mechanisms.

1 Introduction

In the classical—and today still prevailing—applications of business and administration, databases are primarily thought of as *passive* repositories of more or less vast amounts of information that should be available at the fingertips of human clerks, specialists or decision makers. It is these humans that react to outside stimuli, observed situations, orders, or external events to swing into action and, in turn, take the initiative in accessing their databases. Even the more recent object-oriented techniques address issues that still are inherently passive. Their emphasis is, by providing operations and encapsulation, on easing application design, consistency maintenance, and implementation flexibility. By adding strong typing, type hierarchies and inheritance, they also

*This work was supported by the German Research Council (DFG) under contracts Ke 401/6–1 and SFB 346.

achieve a more modular system construction, better adaption to changing environments, and a more economic programming style.

Over the past few years, it has been recognized that at least in the world of office automation, production, or inventory control, the premise of *passivity* is no longer adequate. Actions are not solely stimulated by outside events but should also be due to situations that can be formulated beforehand, e.g., those that exhibit a regular or prespecified temporal pattern or describe distinguished states of the miniworld. For example, in inventory control, the stockroom clerk may be informed automatically whenever the quantity on hand of an article falls below a prescribed limit. If one manages to have an up-to-date and complete image of the miniworld as a database, one may leave the automatic notifications to the database. Databases with these capabilities are often referred to as *active* DBMSs. The prevailing language constructs for active databases are the event-trigger concept and the event-condition-action rules. In these approaches, the events and conditions are specified in declarative form and globally added to the database. This, actually, does not conform with the object-oriented paradigm which, by means of object *encapsulation*, localizes all phenomena.

A different—and more natural—approach for an object-oriented environment is proposed here. We make the individual objects *autonomous* "beings" enhanced with *active behavior*. That is, we provide an object with facilities to go, after recognizing events and exceptional states, into action all by itself, often in parallel with others and perhaps independent of others. For example, the visual system of a robot may recognize the geometry and position of an oncoming workpiece and send the corresponding message to the informational object controlling the robot. The informational control object will then retrieve the corresponding manufacturing description of the workpiece together with all the information for controlling the robot motions from some other database objects and send it to the robot. It may also inform the following station of the upcoming piece, prompt an automatic vehicle (and also the informational object registering its movements) to move to the stockroom to fetch a part and to transport it to the following station for the next assembly step, and send information on the fact that the workpiece has reached the robot to a control panel for display.

Independent activities of systems traditionally come under the notion of *process*. In other words, to perform independent actions, a database system must be triggered into establishing processes, perhaps more than one at a time.

The remainder of the paper discusses which features should be added to traditional object-oriented database environments in order to facilitate autonomy of objects. For the purpose of this discussion, we use GOM, a strongly-typed object-oriented database language. Our previous discussion suggests that such databases should combine three concepts, *object-orientation, active behavior,* and *process* into a single coherent concept.

2 Autonomous Objects

Object-orientation forms the basic premise of our approach, and *process* is the necessary ingredient for carrying activities. *Activities* themselves usually are—as the examples indicated—joint undertakings by several objects cooperating towards a common objective. They require certain *active behaviors* on the part of each object. Accordingly, GOM provides the following support in a homogeneous and coherent framework:

Objects We follow the usual conventions for the notion of object. Its essential features are identified as: *object identity*, *object sharing*, the association of a specific *structure* and, possibly, a set of operators (the *behavior*) for accessing and manipulating the structure, the *classification* of structurally and behaviorally similar objects in types or classes, the *subtyping* of types in order to inherit or refine their structure or their behavior, and the *encapsulation* of objects (Leveson, 1980, Paolini, 1981). We have developed an object oriented database system, called *passive GOM* that provides the above detailed features. Particular emphasis in the design of (passive) GOM was placed on strong typing (Kemper, et al., 1991, Kemper & Moerkotte, 1990c) and optimization (Kemper & Moerkotte, 1990a, Kemper, Moerkotte, & Peithner, 1993).

The prevailing approaches to object-orientation assume that an object remains entirely inactive until one of its associated operations is explicitly invoked by some other client object. It then becomes active and immediately executes the request. After completion it becomes inactive again. Thus, there exists only a single-thread-of-control with its associated master-slave relationship within a collection of passive objects. It all boils down to an invocation mechanism equivalent to the procedure call.

Active Behavior We call *active behavior* an object's potential to perceive and react on special situations. These are

- messages that arrive from other sources,
- passing of predetermined time points, e.g., elapsed time intervals or wake-up calls from clocks, and
- detection of distinguished states or state transitions.

Often these situations are subsumed under the general term of *event*. Note, however, that both the mode of perception and the actions to be taken may differ in the three cases.

In essence, the active behavior of an object defines the range of actions the object may take locally on its own. Associating active behavior with objects necessitates a number of restrictions. Due to encapsulation, monitoring can involve only states internal to an object. Further, it makes sense to associate the perception mechanism with the entire object and let the object then decide which operation to perform. This mechanism, then, appears to guard the entrance to the object, hence we shall refer to it as a *guard* (the similarity to Dijkstra's guards (Dijkstra, 1975) is no accident).

A system of active objects clearly does away with single-thread-of-control because each object may come to life independently of what goes on elsewhere. However, one could still impose a rather strict control regimen by limiting an object to notifying some central agency of its preparedness to act, and to leave it to the agency to schedule its action.

Processes A process is a meaningful sequence of elementary actions. It consumes time and consumes or occupies resources, and it has an observable effect. Such an effect may include changes to some state, and/or messages that are sent off, e.g., to provoke action elsewhere. One can distinguish several process states. Usually four states are identified: created, active, suspended, and completed.

Basically, processes operate entirely independently of one another, i.e., they are subject to truly multiple-threads-of-control. In particular, processes may execute concurrently.

To combine processes and objects, two significant alternatives exist. One, there is just one single master process for each object. Instead, one process is associated with each operation of an object. The former seems more in tune with the object philosophy because treats the object as an entity, thus allowing an object to retain full control over its operations. The latter, however, would allow an object to respond to several stimuli concurrently. Concurrency within one object could be realized by several lightweight slave processes associated with the one heavyweight master process.

Control Activities Interaction in an object world seems to preclude cooperation via shared data items because of encapsulation. This leaves communication as the sole mechanism for interaction of objects. In turn, this requires to set up communication channels between objects. If the communication has as its purpose a collaboration of a certain duration, several messages may have to be exchanged between the participating objects, and these messages may have to follow a certain communication protocol much in the way known from telecommunications. The protocol may have to be enforced by special objects called *(control) activities* (Liu & Meersman, 1992, Lockemann & Walter, 1993).

Autonomy Roughly speaking, autonomy of an object means that the object has full control over its destiny. Control entails a certain degree of decision-making during a process. These decisions concern for one the reaction to observed or received stimuli, for another the actions to be subsequently taken. For a further characterization of the notion of autonomy see (Kemper, et al., 1990).

Control over stimuli entails guards with a certain decision making capability. Typical decisions are to select which stimuli are of interest, to accept or ignore stimuli, to process them in an order determined by the object itself, to delay a response, and to batch the processing of several stimuli. We assume, however, that decision making is not haphazard but follows a given strategy which we refer to as the computation model of the object.

Control over actions subsumes control over the state at any given time. Consequently, an object must be able to decide on its own which state transition to apply and when. This could be seen as a further argument in favor of a single process per object. Quite clearly, autonomy requires a permanent existence of such a process, hence it should be created and started on creation of the object. If a master process suspends itself, its object must include a mechanism for resuming it.

To summarize, an autonomous object has all the qualities of a traditional object, constitutes a process and possesses a set of guards which perceive stimuli and prompt reactions according to some computational model. Consequently, the description of an autonomous object includes the descriptions typical for objects, and additionally a description of the guards which we refer to in the remainder as the *behavioral map* of the object. This concept was strongly influenced by (Andrews, 1981). Autonomous objects were first proposed in (Nierstraz, 1987) and (Tsichritzis, et al., 1987, Casais, 1988) which itself are influenced by the design of the ACTORS language (Agha, 1986). Another implemented approach to active objects is described in (Costa, et al., 1989).

Our current approach to active objects is to view an autonomous object as a strictly sequential process, that is, to have just one master process which allows the processing of only one action at a time. We do not currently support intra-object parallelism. Furthermore, we limit our discussion here to one fixed computational model associated with an autonomous object. This model is based on the nondeterministic selection of the guards in the behavioral map for evaluation. The nondeterministic selection process of the guards may be tuned by associating relative priorities with the guards.

3 The Generic Object Model GOM

In this section we describe the basic concepts of our object model GOM—for now, without the extensions for autonomous objects. The basic GOM concepts are very similar to the essential features identified in the Manifesto (Atkinson, et al., 1989).

We will illustrate our discussion on a drastically simplified logistics control application which, subsequently, serves as a basis to discuss the shortcomings of a passive object model in more detail. In subsequent sections we will introduce the extensions for autonomous objects and then revise the logistics control application in order to achieve a more natural modeling using autonomous objects.

3.1 Overview of GOM

GOM is an object-oriented data model that allows the database designer to model persistent application-specific objects which incorporate their *structural* and *behavioral* description. Like most other object-oriented models GOM allows to classify objects and define a template, called *type*, from which individual objects can be created by instantiation. Additionally, there exists a subtype relation together with inheritance which, however, is not further discussed in this paper.

The structural description of a type can be either one of the following: A *tuple* consists of a collection of typed attributes. The tuple constructor is denoted as $[a_1 : t_1, \ldots, a_n : t_n]$ for unique attribute names a_i and type names t_i. A *set* is denoted as $\{t\}$ where t is a type name. A set of this type may only contain elements of type t or subtypes thereof. A *list* is denoted as $< t >$. Lists are analogous to sets except that an order is imposed upon the elements and duplicate elements are possible.

A simple tuple structured object type *Bolt* is sketched below (the keyword **persistent** initiates the inclusion of the type into the database schema):

```
persistent type Bolt supertype WorkPiece is
    body [Length: float, Thickness: float]
    operations
        declare weight: → float code weightCode; ...
    implementation
        define weightCode is ...
end type Bolt;
```

In this definition we assume that the supertype *WorkPiece* has been defined elsewhere. All of its attributes and operations are inherited by the type *Bolt*. Additional operations are incorporated into the object type *Bolt* by specifying the signature in the **operations** clause and providing the implementation (possibly under separate name) in the **implementation** clause. In GOM it is also possible to *refine* inherited operations.

An example of a collection-valued type is *BoltSet* which is defined below:

```
persistent type BoltSet is
    body { Bolt }
    operations ...
    implementation ...
end type BoltSet
```

GOM provides a large number of built-in operations on collection types. Among the operations on sets are: *cardinality*; *union*, to combine two compatible set-valued objects; *retrieveElems*, to remove a specified number of elements from the argument set and return them as a new set of the corresponding type; *insert* to add a new element to the set, and *delete* to to delete an element from the set.

3.2 The Logistics Control Application

Having sketched the GOM object model we can now attempt a first implementation of a very simplistic logistics control application. We will merely outline an object type called *BoltBox* which responds to two operations:

- *consume* to retrieve a specified quantity of *Bolts* from the *BoltBox* object and
- *supply* to furnish a *BoltBox* object with new *Bolts*.

In this respect instances of the *BoltBox* object type and objects retrieving bolts from them are comparable to the consumer-producer example which is one of the archetypes of a process communication pattern (Andrews, 1981).

The example application is sketched as follows (the keyword **self** refers to the object instance in which the particular operation is invoked):

```
persistent type BoltBox is
    body [MinQOH: int; Box: BoltSet;]
    operations
        declare consume: int → BoltSet code consumeCode;
        declare supply: BoltSet → void code supplyCode;
        declare runninglow: → BOOL code runninglowCode;
    implementation
        define consumeCode(quantity) is begin
            if (quantity ≤ self.Box.cardinality)
                return self.Box.retrieveElems(quantity);
            else ...   !! error: not enough Bolts available
```

```
        end define consumeCode;
        define supplyCode(Bolts) is self.Box.union(Bolts);
        define runninglowCode is return self.Box.cardinality ≤ MinQOH;
    end type BoltBox;
```

The semantics of the above types will briefly be outlined. *BoltBox* encapsulates two attributes: *Box* which serves as a container for *Bolts* and *MinQOH* which is the threshold value at which new *Bolts* should be reordered. Bolts can be retrieved from an instance of *BoltBox* by invoking the operation *consume*. Producers can *supply* new *Bolts* by invoking the operations *supply* with the appropriate argument of type *BoltSet*. In order to enable other objects to check whether the bolts of a *BoltBox* instance are running low the respective operation *runninglow* is declared and defined. This test may be performed, e.g., from time to time or at each invocation of *consume*.

The above model of the *BoltBox*, which is a highly shared object in our (envisioned) logistics control system, bears several severe disadvantages which will be highlighted in the following discussion.

In real-life applications there would, of course, be more than one consumer and/or supplier for the same *BoltBox* instance. These objects may be distributed within the factory and typically work in parallel. This necessitates—under the conventional object paradigm as used in the above programmed example—a centralized scheduler, which is sometimes called the application *script.* The task of this script is to transform the multitude of parallel activities—e.g., *consume* by many consumers, *supply* by several producers, monitoring the object base state to detect abnormal situations—into the single-thread-of-control execution model. As in (Kemper, et al., 1990) we argue that the realization of this centralized, global application script becomes unmanageable in real-life computer integrated manufacturing (CIM) applications—as well as in other complex and highly distributed applications.

Furthermore, the realization of sophisticated, cooperative behavior of objects in response to requests from other objects is difficult. An example may illustrate this. A large *consume* request that cannot be fulfilled due to lack of *Bolts* should be postponed until a new supply arrives. However, in the meantime small *consume* requests should be granted. An autonomous *BoltBox* could have postponed the large *consume* request in order to wait for new incoming supply. There is no need that a consumer should have noticed this autonomous decision of the *BoltBox* object. Under single threat-of-control, this has to be explicitly programmed into the application script.

One of the hardest problems in developing large software products is the integration of several complex subsystems into a single system. Under single-thread-of-control, the integration of two, independently realized subsystems means to develop out of two complex application scripts a single, global new one. The complexity of this task is one of the main reasons that up to now, despite enormous efforts undertaken, there still does not exist an integrated CIM solution. The same argument applies if a new complex subsystem has to be added to a set of existing subsystems. Thus extensibility-in-the-large remains a severe problem even though one of the rationales for using object-oriented systems is the ease with which they may be extended. But this extensibility can, at best, be termed extensibility-in-the-small; it does not provide the much needed support for configuring large object-oriented systems.

4 Autonomous GOM Objects and the Map

Clearly, in order to account for autonomous behavior we should abandon the master-slave relationship incurred by procedure invocation. Therefore, we introduced true message passing in the sense that the receiver is autonomous, i.e., it can determine entirely on its own when and how to react to a message. On the sender side we have to introduce the ability of sending a message, and on the receiver side we have to facilitate buffering and interpretation of incoming messages. To account for object autonomy, a construct called **behavioral map** is introduced. The behavioral map allows to specify the object's autonomous behavior, i.e., the actions to be taken depending on

1. the pending requests for executing operations from the outside world, e.g., other objects or database users,
2. time events, e.g., certain operations that are invoked on a periodical basis, and
3. the current state of the object.

Even though the behavioral pattern of an object could be influenced from the outside environment by received messages it is not—as under the procedure invocation paradigm—completely dictated by the incoming messages. It is the object's autonomous decision if and when it is willing to respond to an incoming message.

Thus, the behavioral map may be considered as the *sensory system* (Kemper, et al., 1990) of the object, which perceives the relevant environmental changes as well as the internal state changes of the object, and initiates corresponding actions, i.e., operations which are declared in the **operations** part of the type definition.

4.1 The Basic Communication Concepts

Before discussing the computational model of the behavioral map we describe the basic communication primitives—discussed here from the sender's point of view. The receiver's reaction upon received messages is subsumed by the behavioral map (cf. Section 4.2) since it is only one of the possible stimuli to which an autonomous object reacts.

A message is a string, in the following referred to as ⟨msg_name⟩, associated with an arbitrary number of arguments. A message is only meaningful if the receiver has an appropriate guard to handle the received message. A message can be relayed to an object using two different message passing paradigms. We have *synchronous message passing* (**call**) and *asynchronous message passing* (**send**). The synchronous message passing makes only sense if the message requests the invocation of some operation within the receiver. In this case the sender (caller) of the message waits until the receiver finishes processing the requested operation. The result, if any, of the operation invocation is returned to the sender and may be assigned to some properly typed variable within the sender object (variable r below). The syntax for synchronous message passing is:

[r :=] **call** ⟨receiver object⟩ ⟨msg_name⟩$(e_1, \ldots, e_n)$

The synchronous message passing involves the blocking of the caller; however it does not—like in procedure invocation—warrant that the callee immediately answers the **call**. This would definitely violate object autonomy, since it involves an outside dictate on the callee's behavior. Rather, the callee may decide to delay the call for some reason, e.g., because some other, more important, stimuli are pending.

Any operation that can be invoked synchronously may, alternatively, be requested by asynchronous message passing. In this case, the sender does not have to wait until control is returned from the receiver. After initiating the **send** the sender can immediately resume its activity and work on other issues. If the requested operation has a return value other than of type **void**, then two possibilities exist:

1. The return value is, by default, discarded by the ⟨receiver object⟩, that is, the sender will never be aware of the outcome of the operation. It will not even know whether the operation was successfully executed at all.
2. The sender may optionally specify some message name via which the return value can be relayed back to the sender by the receiver. Again, this message name ⟨msg_name′⟩ can be any string. But of course, the sender object must now contain some matching guard in order to accept the result.

The syntactic construct for asynchronous message passing is:

send ⟨receiver object⟩ ⟨msg_name⟩($e_1, \ldots, e_n$) [**result via** ⟨msg_name′⟩]

The **result via** clause is, as described above, optional. Only if it is stated then the result of the requested operation will be returned via an implicit **send** of a message called ⟨msg_name′⟩ initiated by the ⟨receiver object⟩ after executing the requested operation. Otherwise, the result is discarded and only the side effects of the executed operation will materialize in the receiver.

4.2 The Behavioral Map

For autonomous objects the type definition of an object is augmented by a **behavioral map** part. Consequently the complete type definition template then looks as shown in Figure 1.

[**persistent**] **type** ⟨type name⟩ **supertype** ⟨type name⟩ **is**
    body ...
    operations ...
    behavioral map
        $\langle\text{guard}\rangle_1$ [**priority** $\langle\text{prio}\rangle_1$] **do** $\langle\text{action}\rangle_1$
            ...
        $\langle\text{guard}\rangle_i$ [**priority** $\langle\text{prio}\rangle_i$] **do** $\langle\text{action}\rangle_i$
            ...
        $\langle\text{guard}\rangle_n$ [**priority** $\langle\text{prio}\rangle_n$] **do** $\langle\text{action}\rangle_n$
    implementation ...
end type ⟨type name⟩;

Figure 1: Template of Autonomous Object Types

The **behavioral map** is a collection of guarded commands, similar to those proposed by Dijkstra (Dijkstra, 1975). More specifically, we distinguish three types of guards:

1. ⟨**on** guard⟩s, which handle requests in the form of received messages,
2. ⟨**at** guard⟩s, which handle time triggered events, and
3. ⟨**if** guard⟩s, which initiate activities dependent on the detection of certain distinguished states.

As mentioned before, we currently support one fixed computational model for autonomous objects which will be discussed below. Note that the model defines the semantics of the behavioral map, not reflect its actual implementation.

As outlined in the Introduction autonomous objects can be thought of as sequential processes in the sense that once an operation has been started it cannot be interrupted by incoming requests or state changes. During their whole lifetime, autonomous objects are active in a way that can be expressed by the following infinite loop:

while alive **do begin**
    select $\langle\text{guard}\rangle_i$;
    if $\langle\text{guard}\rangle_i$ *is passable* **then** $\langle\text{action}\rangle_i$;
end while;

Of course, this cannot be a realization because the availability of unlimited resources would be necessary. An actual implementation has to be optimized in such a way that objects can be deactivated in those time periods where no relevant events occur. An efficient realization is beyond the scope of this paper, though.

As can be seen in the pseudo code section above, an autonomous object constantly selects a guard, tests if it is passable and, depending on the result of this test, executes the respective action. We will first describe how a guard is selected for evaluation, and then give a detailed description of the different possible behavioral map entries.

4.2.1 Nondeterministic Selection

A guard is chosen for evaluation nondeterministically. The user may influence the selection of the guards by assigning integer priorities which are used as relative measures to prioritize certain guards over others. The guards are thus nondeterministically selected for evaluation as indicated by the **priority** clauses. If no priority is given then the priority is assumed to be 1. Note, that this implies that the same impassable guard may be chosen twice or more in succession.

More precisely we have the following meaning. Given n entries in the behavioral map—as shown in Figure 1—where the i-th entry is assigned the priority $\langle\text{prio}\rangle_i$ we calculate the sum $P := \sum_{i=1}^{n} \langle\text{prio}\rangle_i$. We further define $P_k := \sum_{i=1}^{k} \langle\text{prio}\rangle_i$ for $1 \leq k \leq n$. The evaluation of the behavioral map can then be refined as follows:

while alive **do begin**
    $c := \text{random}(1,P)$;   !! random between 1 and P.
    for i **with** $P_{i-1} < c \leq P_i$ **do if** $\langle\text{guard}\rangle_i$ *is passable* **then** $\langle\text{action}\rangle_i$;
end while;

persistent type BoltBox **is**

body ...

operations ...

behavioral map

on consume(q) **if** $q \leq 20$ **and** $q \leq$ **self**.Box.cardinality **priority** 9 **do** **self**.consume(q);

on consume(q) **if** $q \leq$ **self**.Box.cardinality **priority** 1 **do** **self**.consume(q);

on supply($items$) **priority** 90 **do** **self**.supply(items);

implementation ...

end type BoltBox;

Figure 2: Behavioral Map of the BoltBox

The execution model can be thought of as playing roulette. The master process of an autonomous object constantly throws a ball in a roulette wheel whose number of slots equals the number of guards in the behavioral map. Each guard, irrespective of type (**on**, **at**, **if**-guard), has an associated slot whose size is determined by its priority relative to the total priorities assigned. If the ball falls into the ith slot the $\langle\text{guard}\rangle_i$ is evaluated. If it is passable (see Section 4.2.2) the associated action is performed; otherwise the ball is thrown again.

Let us demonstrate the selection mechanism on the above introduced logistics control example. The **behavioral map** of the *BoltBox* may look as shown in Figure 2.

In this behavioral map the first guard is chosen 9 times more frequently than the second one, and the third one is chosen 90 (10) times more frequently than the second (first) one. This way, we give very high priority to incoming supplies in order to avoid emptying the storage space by processing requests while incoming supplies are waiting in the message list. Furthermore, incoming requests are carried out under different priorities: small consume requests, i.e., quantities not exceeding 20, are evaluated with priority 9, while all other consume requests—including the small ones—are evaluated with priority 1.

4.2.2 The Guards

In the beginning of this subsection, we introduced three kinds of guards. Let us first discuss the $\langle$**on** guard$\rangle$s which initiate actions as a response to received messages from the outside environment, e.g., requests by other (client) objects or requests by the database user. The full syntax of the $\langle$**on** guard$\rangle$ clause is defined as follows:

$\langle$**on** guard$\rangle$ ::= **on** $\langle$msg_name$\rangle$($p_1, \ldots, p_n$) [**from** $\langle$sender$\rangle$] [**if** $\langle$cond$\rangle$]

All the $\langle$**on** guard$\rangle$s within the behavioral map share a single message buffer as indicated in Figure 3, into which all incoming messages are inserted—in the order in which they are received. If the guard chosen according to Section 4.2.1 happens to be an $\langle$**on** guard$\rangle$ this initiates a search in the message list for the *first* message from the beginning of the list (the bottom in Figure 3) for which the **on** clause including the **from** and the **if** part are satisfied. If one is found, then the **do** part is executed, and the message is deleted from the message list. If no such message is found in the entire message list the system selects another guard for evaluation. It

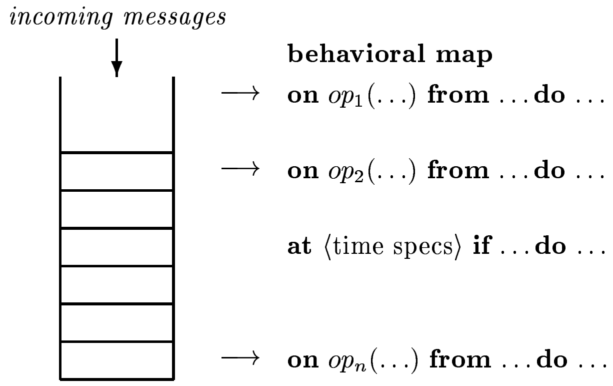

Figure 3: Graphical Representation of the **map**

should be stressed that the message list does not necessarily obey the FIFO (first in first out) order. Requests can be removed from any position in the message list.

For illustration, reconsider the behavioral map of the *BoltBox* as specified in Figure 2. Note that this behavioral map prevents starvation of large *consume* requests because the **on** guards always search for the first qualifying entry within the message list starting at the beginning—and rearranging the message list is not allowed. Eventually, every *consume* request, even a large one, moves to the beginning of the message list where it is then carried out by the second guard.

An ⟨**on** guard⟩ evaluates to true if there is a pending message named ⟨msg_name⟩ with associated n parameters $p_1, \ldots, p_n$ in the common message list. Note that this need not be the message at the bottom of the list. Additionally, the following conditions must hold:

1. if the **from** clause is specified the message ⟨msg_name⟩ must be one received from the specified ⟨sender⟩.

2. the optional **if** condition ⟨cond⟩ evaluates to true. The predicate ⟨cond⟩ may refer itself to the parameters of the message ⟨msg_name⟩.

If all conditions are satisfied the ⟨**on** guard⟩ is said to be *passable*.

Aside from outside requests which are handled by the **on** clauses an autonomous object may also need to respond to time-triggered actions. For this purpose the behavioral map may contain one or more ⟨**at** guard⟩s which have the following syntactical form:

⟨**at** guard⟩ ::= **at** ⟨date⟩ [**every** ⟨duration⟩ [**until** ⟨date⟩]][**if** ⟨cond⟩]

The ⟨**at** guard⟩ is evaluated like the ⟨**on** guard⟩, except that now the search for a matching message is replaced by checking whether the specified time for the next time-triggered event has been reached. Of course, it cannot always be guaranteed that an ⟨**at** guard⟩ is evaluated at exactly the specified time; therefore, the **at** clause evaluates to true if the specified time has been reached and the guard has not been passed since.

The ⟨**if** guard⟩ constitutes the third, and last, kind of guard that our autonomous object model supports. These guards have the very simple syntactical form

⟨**if** guard⟩ ::= **if** ⟨cond⟩

This guard is *passable* if the condition ⟨cond⟩ evaluates to true. This allows the triggering of actions dependent on states of the object. If the **if** entry is missing this can be read as **if** *true* **do** ⟨action⟩.

5 The Logistics Control Example Revisited

In this section we demonstrate how our concepts of autonomous objects and communication between them can be used to model our logistics control system in a manner that conforms directly to the dynamics of the application and thus is a very natural representation.

We now indicate the design of the *BoltBox*, i.e., the shared autonomous GOM object that controls the bolts that are currently in stock.

```
persistent type BoltBox is
    body
        [MinQOH: int; Box: BoltSet;
         Control: LogisticsControl]
    operations
        declare consume: int → BoltSet code consumeCode;
        declare supply: BoltSet → void code supplyCode;
    behavioral map
        on consume(quantity) if quantity ≤ self.Box.cardinality priority 1 do
            self.consume(quantity);
        if self.Box.cardinality < self.MinQOH priority 1 do
            send self.Control RunningLow(self.Box.cardinality);
        on supply(Bolts) priority 10 do begin
            self.supply(Bolts); send self.Control BoltsSupplied;
        end;
    implementation
        define consumeCode(q) is return self.Box.retrieveElems(q);
        define supplyCode(Bolts) self.Box.union(Bolts);
end type BoltBox.
```

We have defined the autonomous *BoltBox* type such that it notifies the logistics control whenever the still available quantity (*Box.cardinality*) is running low. We can now define the associated *LogisticsControl* object which—autonomously—takes care of reordering bolts whenever the stock is running low and of paying those new bolts when it is notified about their supplement.

```
persistent type LogisticsControl is
    body
        [BoltsOrder: int; Box: BoltBox;
         BoltsReordered: BOOL:= FALSE; ObjectsToNotify: {ANY}; ...]
```

```
operations ...
behavioral map
    on RunningLow(quantity) from self.Box if not BoltsReordered do begin
        send CentralPurchasing reorder("Bolts", BoltsOrder − quantity);
        self.BoltsReordered:= TRUE;
        foreach oid in self.ObjectsToNotify do send oid signal;
    end;
    on BoltsSupplied from self.Box do begin
        self.BoltsReordered:= FALSE;
        ...;    !! take care of payment
    end;
    on notify(oid) do self.ObjectsToNotify.insert(oid);
    on unnotify(oid) do self.ObjectsToNotify.delete(oid);
implementation ...
end type LogisticsControl.
```

The *LogisticsControl* object multicasts messages signalling that the bolt box is running low in order to multiplex the flow of control. It would now be very easy to incorporate another agent object that is interested in detecting any shortages of *Bolts*. Objects which are interested in such an event need only send a *notify* message to the *LogisticsControl* object or an *unnotify* message when they are not interested any longer. Assume, for example, that there is some chief (human) operator in the CIM environment who is in charge of controlling the logistics within the system, in order to interfere if anything goes wrong. For this purpose we may define a type *OperatorConsole* which displays any such abnormal state descriptions:

```
persistent type OperatorConsole is
    body
        [Control: LogisticsControl; ...]
    ...
    behavioral map
        on signal from self.Control do display("Running out of bolts!");
    ...
end type OperatorConsole;
```

In the same way as we have shown with the autonomous type *LogisticsControl*, some other kind of *relais* object can be incorporated into the system. This makes it easy to extend existing applications without having to alter the internal implementation of autonomous objects—if it is known beforehand what kind of *relais* are needed and which communications shall be directed to those *relais*. In our example, the operator console could be integrated into the system without augmenting the *BoltBox* or the *LogisticsControl* definition.

Now the question may arise how to deal with constraints involving more than a single autonomous object, possibly of different types. One of our main concerns in this paper is to deal with autonomous, loosely coupled objects—as they are a matter of fact in many "modern" application environments, as e.g., production and office automation. Thus, a global behavioral map would not be a solution that meets these premises. Instead, the answer to this question is cooperation between autonomous objects. In this paper we proposed some basic constructs to specify object autonomy and communication between autonomous objects. Based on these primitives an extended model that gives the application programmer more support in the specification of cooperation, ca be designed.

6 Conclusion

The main hypothesis of this paper has been that autonomous objects which have variously been mentioned in the literature are an ideal means to model the highly complex and dynamic applications of technical environments. It turned out that, before doing so, a more precise meaning has to be given to the notion of autonomous object than has been the case so far. Basically, an autonomous object has been considered an object in the widely accepted sense that is associated with a permanent process which controls its own state transitions. It incorporates a set of guards that accept stimuli from outside or inside the object according to predefined conditions. Its actions follow a predefined computational model.

Precision in the definition of the basic notions and provision of a simple but powerful conceptual framework for systems of objects are essential prerequisites for developing a concise language through which to express the informational model of a given technical environment. Such a language containing autonomous objects offers an abstraction mechanism in order to model subsystems, e.g., subunits of an organization, in an independent manner. The integration of the subsystems can be done without superimposed control structures that tend to be very complex (sometimes too complex) and thus very difficult to realize in the applications considered in this paper. A model implemented with that kind of language can be interpreted as (part of) a program for cooperating processes. Considerable part of the paper has been devoted to the description of such a language and to a demonstration of its utility. Chief among the language constructs has been the concept of a behavioral map.

So far we have carried out only first, non-conclusive studies concerning *synchronization* of activities in our model. Based on the works by Liskov and Weihl (1985) and Weihl (1989), Herlihy (1990), and Schwarz and Spector (1984) it is relatively straightforward to incorporate transactions into our model by making the autonomous object types *atomic*, that is, equip the types participating in transactions with their own localized concurrency control schemes. However, the conventional transaction mechanism turns out too inflexible for long-running CIM applications; therefore, we have to work on more flexible concepts, like *Sagas* (Garcia-Molina, 1987) and *ConTracts* (Reuter, 1991), which should be provided as built-in concepts that can be customized by the user to suit his or her needs. This may be done through control activities (see Section 2).

The autonomous objects we deal with so far are still a bit restricted. The computational model is the same for all objects and there is no intra-object parallelism. Also, the behavioral map is still too static. It should provide more flexibility, e.g., dynamically altering the priorities of the guards, in order to fine-tune the behavior of individual autonomous objects within a complex system.

Also, it seems to be worthwile to extend the **on** clause of the guards not only to contain messages but full event-expressions as proposed, e.g., in (Gehani, et al., 1992). The basic expressions of such an event-definition-language are expressions denoting some communication between objects, i.e., the sending or receiving of a message. Note, that we consider it important, that these events comprise not only communications that are directed to or coming from the object on which the guard is defined but also communications between arbitrary objects. By this, an object is enabled to supervise the communication between a group of objects in order to establish some kind of communication protocol between those objects (Lockemann & Walter, 1993). From these primitive event-expressions, composite events can be constructed in order to gain more control over the communication behavior of autonomous objects.

References

Atkinson, M., Bancilhon, F., DeWitt, D. J., Dittrich, K. R., Maier, D., & Zdonik, S (1989). The object-oriented database system manifesto. *Proc. Int. Conf. on Deductive and Object-Oriented Databases*, (pp. 40–57), Kyoto, Japan.

Agha, G. A. (1986). *ACTORS: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA: The MIT Press.

Andrews, G. R. (1981). Synchronizing resources. *ACM Trans. Programming Languages and Systems*, 3(4), 405–430.

Casais, E. (1988). An object oriented system implementing KNOs. *Proc. Conf. on Office Information Systems (COIS)*(pp. 284–290). Palo Alto.

Costa, J. F., Sernadas, A., & Sernadas, C. (1989). OBL-89 user's manual. Technical report. Instituto Superior Tecnico, Lisbon.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8), 453–457.

Gehani, N. H., Jagadish, H. V., & Shmueli, O. (1992). Composite event specification in active databases: Model and implementation. *Proc. of the 18th Int. Conf. on Very Large Data Bases (VLDB)* (pp. 327–338). Vancouver, British Columbia, Canada.

Garcia-Molina, H., & Salem, K. (1987). Sagas. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data.*

Herlihy, M. (1990). Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Systems*, 15(1), 96–124.

Kemper, A., Lockemann, P. C., Moerkotte, G., Walter, H.-D., & Lang, S. M. (1990). Autonomy over ubiquity: Coping with the complexity of a distributed world. *Proc. Ninth Intl. Conf. on Entity Relationship Approach.* Lausanne, Suisse.

Kemper, A. & Moerkotte, G. (1990a). Access support in object bases. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, (pp. 364–374). Atlantic City, NJ.

Kemper, A. & Moerkotte, G. (1990c). Correcting anomalies of standard inheritance—a constraint based approach. *Proc. Intl. Conf. on Database and Expert Systems Applications*, (pp. 49–55). Wien, Austria.

Kemper, A., Moerkotte, G., & Peithner, K. (1993). A blackboard architecture for query optimization in object bases. *Proc. of The Conf. on Very Large Data Bases (VLDB)*, (pp. 543–554), Dublin, Ireland.

Kemper, A., Moerkotte, G., Walter, H.-D., & Zachmann, A. (1991). GOM: a strongly typed, persistent object model with polymorphism. *Proc. of the German Conf. on Databases in Office, Engineering and Science (BTW)*, (pp. 198–217). Kaiserslautern, Germany: Springer-Verlag.

Leveson, N. G. (1980). Applying Abstract Data Type Methodology to Data Base Systems. Ph.D. Thesis, UCLA. Los Angeles, CA.

Liu, L. & Meersman, R. (1992). Activity model: Declarative approach for capturing communication behaviour in object-oriented databases. *18th International Conference on Very Large Data Bases.* Vancouver, British Columbia.

Lockemann, P. C. & Walter, H.-D. (1993). Activities in object bases. *Proc. of 1st Int. Workshop on Rules in Database Systems.* Edinburgh, UK.

Nierstrasz, O. M. (1987). Active objects in HYBRID. *Proc. of the ACM Conf on Object-Oriented Programming Systems and Languages (OOPSLA).*

Paolini, P. (1981). Abstract Data Types and Data Bases. Ph.D. Thesis, UCLA. Los Angeles, CA.

Reuter, A., Wächter, H. (1991). The contract model. *IEEE Data Engineering Bulletin*, 14(1).

Schwarz, P. M. & Spector, A. Z. (1984). Synchronizing abstract types. *ACM Trans. Computer Systems*, 2(3), 223–250.

Tsichritzis, D., Fiume, E., Gibbs, S., & Nierstrasz, O. (1987). KNOs: Knowledge acquisition, dissemination and manipulation objects. *ACM Trans. Office Information Systems*, 5(1), 96–112.

Weihl, W. E. (1989). Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Programming Languages and Systems*, 11(2), 249–282.

Weihl, W. E. & Liskov, B. (1985). Implementation of resilient, atomic data types. *ACM Trans. Programming Languages and Systems*, 7(2), 244–269.