Using Reflection as a Means of Achieving Cooperation*

David Edmond, Mike Papazoglou and Zahir Tari
School of Information Systems, Queensland University of Technology
GPO Box 2434 Brisbane Queensland 4001 Australia
email: {davee,mikep,zahirt}@icis.qut.edu.au

Abstract

We view cooperating systems as being a set of systems that are distributed over a common communication network and that work towards solving a common task. This is achieved by coordinating and exchanging information and expertise. Such systems exclude conventional database systems since the knowledge these systems contain is buried within application code. In this paper, we overcome this problem by introducing a layer of special reflective, i.e. metalevel, objects which surround each local database system. These objects are used to capture domain and operational knowledge, and to describe, at least in part, remote systems and to monitor task-oriented activities. In this way, we can turn interconnected conventional database systems into a set of cooperating knowledge-based systems.

1 Introduction

How often, when using an information system, do we ask ourselves: "I wonder why it did that?" or "How on earth did that happen?". Unfortunately, the questions are rhetorical. We do not have the answers, and none will be forthcoming from the system. But, surely, the power of any data or knowledge modelling system does not merely depend upon the immediate properties of its representational structures. It also depends upon how well such a system may represent and reason about its own structures and functionality. The capability of self-representation is known as *reflection*, and has been particularly successful in object-oriented settings. There, it has been employed as a novel methodology for constructing flexible, large-scale complex systems such as (1) programming languages [Mae87, Mae88], [KRB91], [SSF92], [MM+92], (2) the developing the next generation of operating systems so that they provide open-ended and self-extending facilities [Yo+91, Yo92] operating systems, and (3) window systems [Rao91].

The major characteristics of a reflective system are:

- The clear separation of *domain* knowledge from *control* knowledge [Dav80].
- The *explicit* representation of that knowledge.

Domain knowledge is *what* a system knows of its domain, and is encoded within the application system. Control knowledge is *how* that knowledge is or should be applied, and is encoded in the metalevel. The KADS methodology [BR+90, BB+91, WSB92] emphasises the importance and complexity of this kind of knowledge by further refining three forms of control knowledge. In a database system, we would say that (roughly speaking) the domain knowledge is encoded by the applications programmer

*This research is partially funded by a grant from the Australian Research Council.

following specifications of end-user knowledge captured by a systems analyst; and the control knowledge is supplied, in a domain-independent way, by the database management system, using *its* knowledge of internal data structures to optimise the retrieval of data. There are two major reasons for explicitly representing control knowledge[vHa91]:

- A system with separate representations is simply easier to build, debug and develop.
- The system is then able to explain why it took the actions it did, why it used domain knowledge in a particular way.

Reflection provides *explicit* mechanisms for expressing user-specified policies and requirements within an object-oriented system. It offers a clean adaptable interface through which users can customize systems according to their requirements. For example, reflection can be used to particularise individual object properties in situations where class-based inheritance is too cumbersome. Object-oriented database systems are rather rigid in their approach to modelling in the sense that they attach to all objects originating from the same class a single set of semantics and properties. These limitations of object-oriented database systems may be overcome by the provision of appropriate metaobjects in addition to the conventional objects. In general, conventional objects may be interpreted as carriers of domain information, whereas metaobjects define the semantics of object actions and overall behaviour. For example, each object may have its own group of metaobjects which provide a set of descriptions and metaoperations that define the object's semantics.

Not only does reflection help in applying domain semantics, it can also be used to provide the means to alter the dynamic, run-time behaviour of a language. For example, the reflective language 3-KRS [Mae87, Mae88] provides a meta-object per object. This meta-object is used to control the execution of messages that are sent to its referent object by dispatching an appropriate method. This facility can be used in situations where two objects originating from the same class may need to respond differently to the same message. CLOS [KRB91] is another object-oriented language that provides self-describing facilities whereby a collection of classes, the meta-object classes, represent all the major building-blocks of the language. The CLOS meta-object classes constitute the core facilities of its meta-object protocol (MOP). The MOP can be used to describe operations and interactions among instances of its meta-object classes in a way that allow extensions or modifications of the CLOS implementation. For example, the PCLOS system [Pae90] has added database persistence to CLOS objects by subtyping meta-object classes and selectively shadowing methods that operate on their instances. These implementation changes come into effect without requiring any modifications to the existing system code of CLOS.

We view cooperating systems as being a set of systems that are distributed over a common communication network and that work towards solving a common task. This is achieved by having systems that coordinate and exchange information and expertise. Such systems typically exclude conventional database systems since the knowledge these systems contain is buried within application code.

The use of reflection is particularly beneficial to multi-database systems or cooperating systems. Cooperative tasks, carried out in an ad-hoc manner up to now, can be performed using reflective data modelling facilities. Such reflective tasks may include transaction scheduling, object communication and method dispatching to remote objects, object synthesis and composition, object migration (especially beneficial for mobile computing purposes [Yo+91]), distributed object implementation policies, etc.

The work reported in this paper is based on the assumption that we have a set of essentially passive and autonomous databases located at a number of remote sites. To enable some form of cooperation between these sites, we extend their functionality by introducing a layer of metalevel software that

surrounds each local database system. This metalevel software knows about each site's capabilities and functionality and consists of:

- A complete metadata description, in terms of system competence.
- Metaobjects that provide basic cooperative processing facilities.

These metaobjects are designed to capture domain and operational knowledge, and to describe, at least in part, remote systems and to monitor task-oriented activities. The term *meta* object is used only to indicate the relation of such an object to the object it describes. A metaobject is just another object, with structure and behaviour. However, it too has access to descriptions of itself.

In this way, we can turn interconnected conventional database systems into a set of cooperating knowledge-based systems. Reflection not only allows descriptions of the capabilities of existing information systems and their inter-relationships but also facilitates the specification and implementation of a newly composed system, by drawing upon the functionality of these already existing systems.

In this paper we introduce aspects of the R-OK Model – a generic object model to support cooperative processing activities – where every object has access to four metaobjects which, together, provide the capabilities outlined previously. The paper is structured as follows: in section 2, we discuss the basic ideas or constructs of R-OK; in section 3 we use these constructs to penetrate aspects of information systems that are usually closed to us; in section 4, we look at two examples of how knowledge of behind-the-scene actions may be used to enable cooperation; in section 5, we make some conclusions.

2 The R-OK Model

In this section, we introduce the reflective aspects of the model. In particular, we examine a group of four special kinds of objects that may be used to describe and monitor *other objects*. These *reflective* objects provide access to *metalevel* aspects of an information system that are often hidden. Using reflection in this manner, this metalevel is revealed in object-oriented terms. To provide straightforward explanatory examples, we will examine a database that is used by a diagnostic department within a hospital. The system consists of a set of `Patient` objects. The current state of a typical patient object, `Jim`, is shown in Figure 1. For each such object, such as `Jim`, we have the metalevel objects of the kinds discussed in the following subsections.

2.1 The `state` metaobjects

These are objects that know the structure of any associated object, whether that object be a domain or a metalevel object. According to Figure 2, the `state` metaobject is an object with two attributes: (1) *State* which is a record type or scheme, and (2) *Context* which allows a name to be given to the record type. Thus, `state(Jim)` is an object that knows about the structure or state of `Jim`, that is, its attributes and their types.

The `state` metaobject shown in Figure 2 is one that is probably, but not necessarily, shared by several domain objects. In using the term "sharing", we mean that there may be a one-to-many relationship between a given `state` metaobject and the domain objects that it describes. It is the very existence of a `state` object with a context called `Patient` that allows us to refer to `Patient` objects. In doing so, we are making use of a shareable metaobject. The methods associated with a `state` object will allow the addition, modification, and removal of the attributes described in the *State* component of the metaobject.

A Patient object	
Attribute	*Value*
PatientId	12875
Name	Jim
Ward	12
DOB	23-Jan-1965
Drugs	{}
Allergies	{aspirin,albumen}

Figure 1: A snapshot

The state *metaobject*

Context: Patient
State:

Attribute	*Type*
[PatientId:	Integer;
Name:	CharString;
Ward:	CharString;
DOB:	Date;
Drugs:	Set CharString;
Allergies:	Set CharString]

Figure 2: The state metaobject for a patient

2.2 The can metaobjects

These objects know about the behaviour of any associated object – they know what it *can* do. From Figure 3, we see that the can metaobject also has two attributes. It has an attribute *CanDo* which pairs method names with their specifications. The attribute *Context* allows a name to be given to this particular set of pairings. There are three methods in this example, `MakeBooking`, `AddTreatment` and `MoveWard`. The specification of the `AddTreatment` operation is presented in terms of four components:

- `requires` which contains the argument(s) that will be supplied. In this case, there is one argument, drug which must be a character string.
- `reply` which says what kind of information, if any, will be returned. In this case, there is none.
- `pre` which is a precondition for the method. In this case, the drug must not be one to which the patient is allergic.
- `post` which describes *the effect* of the operation upon the object. In this case, the drug is added to the treatment set out for the patient. The convention used is that attributes with a subscript, eg $\texttt{Drugs}_0$, represent the value of the attribute *before* the operation. Unsubscripted attributes refer to the value *after* the operation is finished.

The can *metaobject*

Context: Patient
CanDo:

Name	*methodSpec*	
MakeBooking	...	
AddTreatment	requires:	[drug: CharString]
	reply:	[ ]
	pre:	drug not in Allergies
	post:	Drugs = Drugs_0 union {drug}
MoveWard	...	

Figure 3: The can metaobject for a patient

`can(Jim)` is an object that knows about Jim's possible behaviour, that is, what messages it might be sent and the structure or pattern associated with each. In general, a can object is goal- or ends-oriented.

It describes the effect that each of the methods should have on the object concerned but not *how* this effect is achieved. Like all four of these metaobjects, the can metaobject is one that may be shared by several domain objects. In this case they would be objects that exhibit the same behaviour. The methods associated with a can object will allow (1) the addition of new behaviour to the corresponding domain objects, and (2) the refinement of existing behaviour through the alteration of either pre- or post-conditions.

2.3 The act metaobjects

These objects know about the *current* activity surrounding other objects. act(Jim) is an object that knows about Jim's current activity, that is, messages it is still processing, and messages it has despatched and for which it is awaiting a response. It is possible for two or more objects to share the same act metaobject.

The act *metaobject*

Context: Patient

In:

from	*for*	*action*	*with*	*place*
ProgA	Karl	MoveBed	bed is B100	1
ProgC	Mary	AddTreatment	drug is penicillin	2
ProgE	Angus	AddTreatment	drug is TLC	3

Pending:

from	*for*	*action*	*with*
ProgB	Jim	AddTreatment	drug is aspirin
ProgD	June	MoveWard	Ward is 99

Figure 4: The act metaobject for patients

According to Figure 4, there messages have come in for Karl, Mary and Angus. None of these have been processed yet.

The act metaobject is task-oriented. Its job is to supervise the activity of the one or more objects within *its* domain. These objects may or may not be of the same kind, that is, ones with the same structure and behaviour. In the example given, they are, but more generally, an act metaobject may supervise a heterogeneous set of objects that are collectively attempting to perform some task. For example, in a building control system, we might have objects representing the lighting level and objects representing the air-conditioning level being monitored by the same act metaobject. In general, for an object X to have an act(X) means that X delegates responsibility for (1) the validation of a message, (2) the timing of its execution, and (3) possibly the timing of any reply to that message. For this reason, the act object will need to be aware of the timing of events such as the receipt and despatch of messages. We have omitted such discussion from this paper.

2.4 The loc metaobjects

These objects know where to go to *locate* attributes and/or methods. They answer the question: Do you really want to know how its done or where to find it? They allow the particular state of an object, at any time, to be materialised; and they allow its methods to be executed. A loc metaobject contains two attributes. *Lookup* is a function that maps from a name to an object, one that may possibly reside at

some remote site. In Figure 5, each patient attribute is paired with an object that will handle requests to retrieve or update the associated attribute.

The *Do* attribute maps from a name to a procedure – one that should correctly implement the corresponding method specification in the `can(Jim)` object. It might be thought that the distinction between attributes and methods has been carried into this object – and so, perhaps, it should be split in two. However, for some objects, there might be methods that are handled through remote procedure calls – in which case, they would appear in *Lookup*. Conversely, there might be attributes that are derived procedurally, in which case, they would appear in the *Do* function.

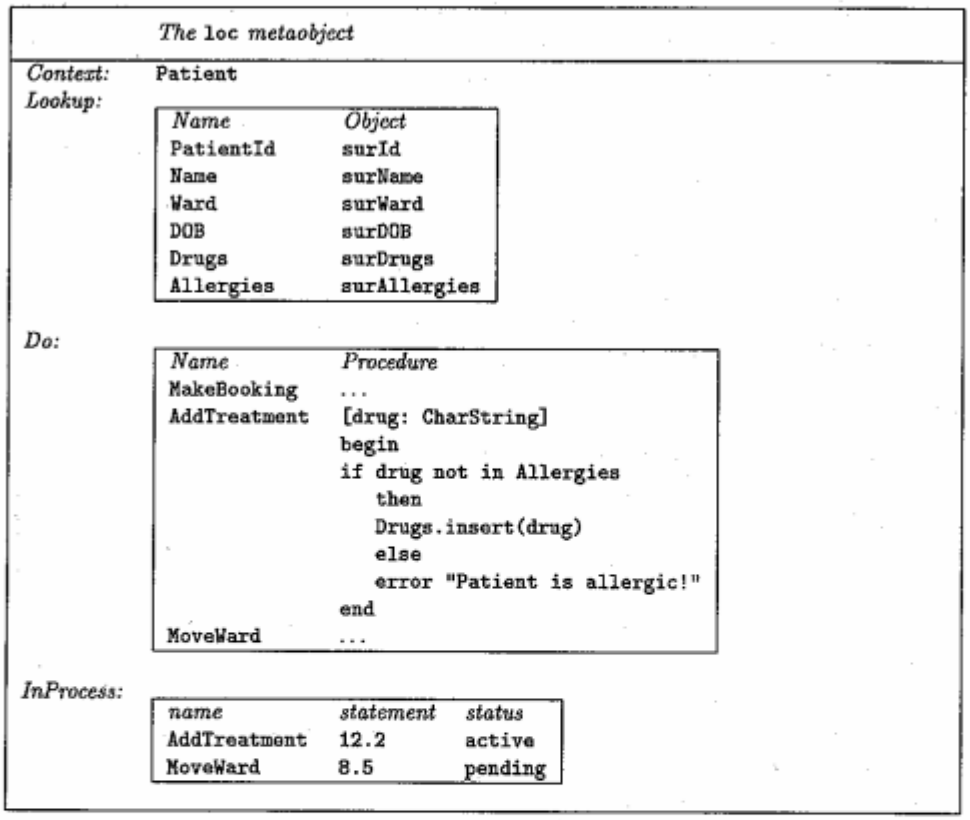

Figure 5: The `loc` metaobject for a patient

When an interpreter, acting as one of the methods of a `loc` object, encounters an unrecognised symbol, it looks up the symbol table provided, ie the *LookUp* attribute, for assistance. If the symbol appears on the right-hand side of an assignment statement then a `get` message is sent to the associated **surrogate** or **lookup** object. If the symbol appears on the left-hand side, then a `set` message is sent. Obviously the interpreter executes the `get`(s) before the `set`. In its simplest form, `loc(Jim)` is an object that knows how to synthesize the attributes and how to locate the methods. An attribute is materialised by sending an appropriate message to some predetermined surrogate object and waiting for a response.

Suppose that there is a relational database that is "substantiating" the `Patient` objects. The set-valued `Allergies` attribute is stored as a two-column table `Allergies(PatientId,Allergy)`. According to the `loc(Patient)` object, the `Allergies` attribute is mapped, through the *Lookup* function, to a

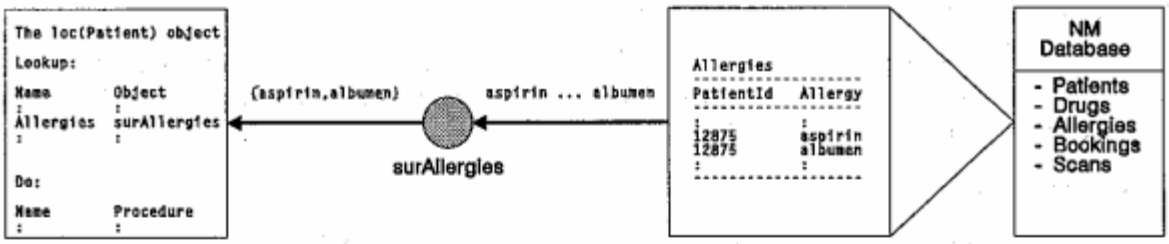

Figure 6: Locating and retrieving allergies

surrogate object `surAllergies`. This object, indicated as a shaded circle in Figure 6, is the object that will deal with any requests to access the patient's allergies. If it has been charged with retrieving patient `Jim`'s allergies, then it will select these from the `Allergies` table using SQL, gathering them individually using a cursor; then it will marshall the results and pass them back, as a set, to the `loc(Patient)` object.

3 Further behind the scenes

The `state` and `can` metaobjects describe the structure and behaviour of `Jim`, where `Jim` is considered as a *semantic* object, that is, one that corresponds to an object in the application domain. However, this conceptual object may be composed by synthesising other pre-existing objects. The `loc` and `act` objects describe the structure and behaviour of the implementation of `Jim`, where `Jim` is now considered as an *implementation* object.

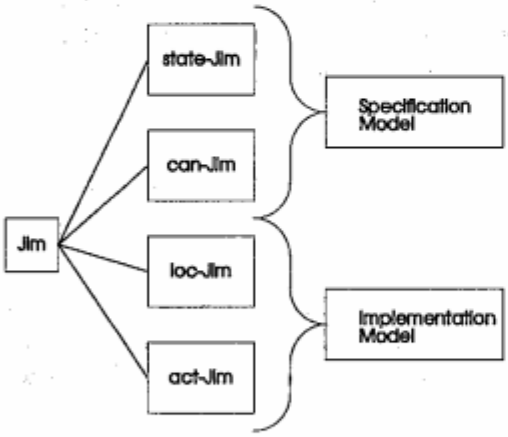

Figure 7: Meta-Jim

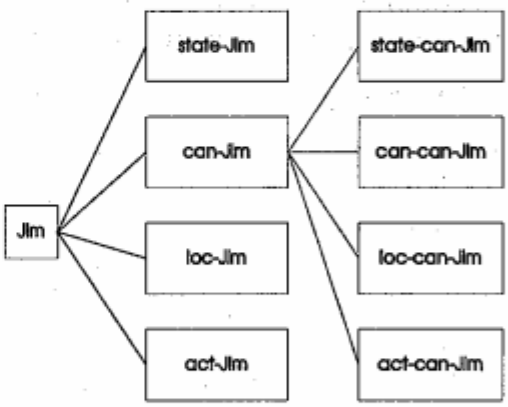

Figure 8: Meta-Meta-Jim

We have used the metaobject mappings to investigate the domain objects. According to Figure 7, there are four metaobjects associated directly with `Jim`. But metaobjects may also have their metaobjects. There are sixteen possible metaobjects at the next level. Figure 8 shows four of them. See [EPT95] for a more detailed discussion. In this section, we examine two aspects and relate them to cooperation.

3.1 Instantiation and Materialisation

The four objects represent a gateway from any object into the metalevel. Each of them represents a different act of reflection. What is required to make an object? In a typical object-oriented language, we could make a declaration like the following:

```
p:  Patient
```

This not only declares the structure and behaviour of the object p, it implicitly declares how and where the object is to be constructed and located. In a C++ program, for example, a suitable chunk of memory is allocated from the heap and initialised appropriately.

In R-OK, we may make a declaration like the one above, but in doing so, we are declaring the particular metaobjects to be used, each of which plays a role. Through this declaration, we are saying that:

- The object has attributes of the kind specified in the `state` object with `Context = "Patient"`.
- Its behaviour by referencing a `can` object with context `"Patient"`.
- Its attributes may be located, and its methods executed, in the way found in the `loc` object with context `"Patient"`.
- It is to be monitored by an `act` object with the context `"Patient"`.

These latter two aspects of a declaration differentiate R-OK from more conventional systems. They are essential for wrapping pre-existing application systems. We must able to state how the objects are implemented, and to say how these objects are monitored.

Alternatively, we may refine that declaration. Suppose we have another patient database that we want to merge with the existing one. We provide a suitable `loc` metaobject which maps to this new database, one with a context `NewSource` say. We may declare a patient object as follows:

```
p:  Patient except loc(p).Context=NewSource
```

The object will share all the features of `Patient` objects except that it is located somewhere else.

3.2 Message Processing

Message processing is closely related to the activities of the `act` objects. Their behaviour is specified in the `can(act)` object shown in Figure 9. This object describes the behaviour of the `act` object which receives messages, validates them and, at a time of its own choosing, despatches them. A message consists of the following basic components, examples of which may be found in Figure 4:

- `From:` identifies the source of the message.
- `For:` identifies the destination or target object.
- `Action:` is the action to be taken in regard to the previous object.
- `With:` contains any arguments or information supplied.

The steps in passing a message `M` from object `A` to object `B` are as follows:

- The interpreter, which is a method of `loc(A)`, encounters a request to send a message `M` to object `B`. It will find such a request within the body of some procedure associated with `A`, and will send a message to `act(B)` asking it to deal with `M`.
- `act(B)` receives this message, stores it, and eventually despatches it to `loc(B)`.
- `loc(B)` receives this message, locates the corresponding procedure and commences executing that code. Alternatively, it may find somewhere to forward the message.
- As the interpreter works through the code, it uses the *Lookup* attribute of `loc(B)` to to determine how to handle unknown symbols that it encounters. Typically, these will be attribute names, and there will be a surrogate object associated with that attribute.
- `loc(B)` sends messages to these surrogates, asking them to `get` or `set` the value of the corresponding attribute.
- Once `loc(B)` has finished, it will return any reply to message `M` to `act(B)` which will, in turn, pass it back to `loc(A)`.

Here is an example of a simple message for `Jim`:

```
From        For       Action         With
Prog        Jim       AddTreatment   [drug is aspirin]
```

where `Prog` is some program. The hidden message, so to speak, is slightly different. It is as follows:

```
From        For       Action         With
loc(Prog)   act(Jim)  Receive        [from is Prog, for is Jim,
                                      action is AddTreatment,
                                      with is [drug is aspirin]]
```

The message comes from `loc(Prog)` because it is there that the program code is interpreted. The target object is `act(Jim)` because that object receives messages on behalf of `Jim`. The action is `receive`, and the argument is the whole of the original message.

An `act` object may take the following actions (see Figure 9):

- `Receive`: it may receive a message `msg`. This will be one that wraps some other message in the way shown in the above example, ie the component `msg.With` will itself be another message. The precondition is that the action specified in that inner message, `msg.With.Action`, is one that is named in the `CanDo` attribute of the `can` object of the object for which the message was originally targetted, `msg.With.For`. The `CanDo` attribute is a relationship, in the form of a set of pairs. The set expression involved in the precondition is of the form:

  `{x: Set •Exp}`

  This lets `x` range over the set `Set` and forms a set based on the value of the expression `Exp`. In this case we have `{m: can(msg.With.For).CanDo•m.Name}` which forms a set consisting of the names of message actions that the target object will handle. The postcondition is:

  `In = In₀^<msg.With>`

  This requires that the original message be turned into a singleton sequence, `<msg.With>`, and added (`^`) to the end of the input queue, In_0.

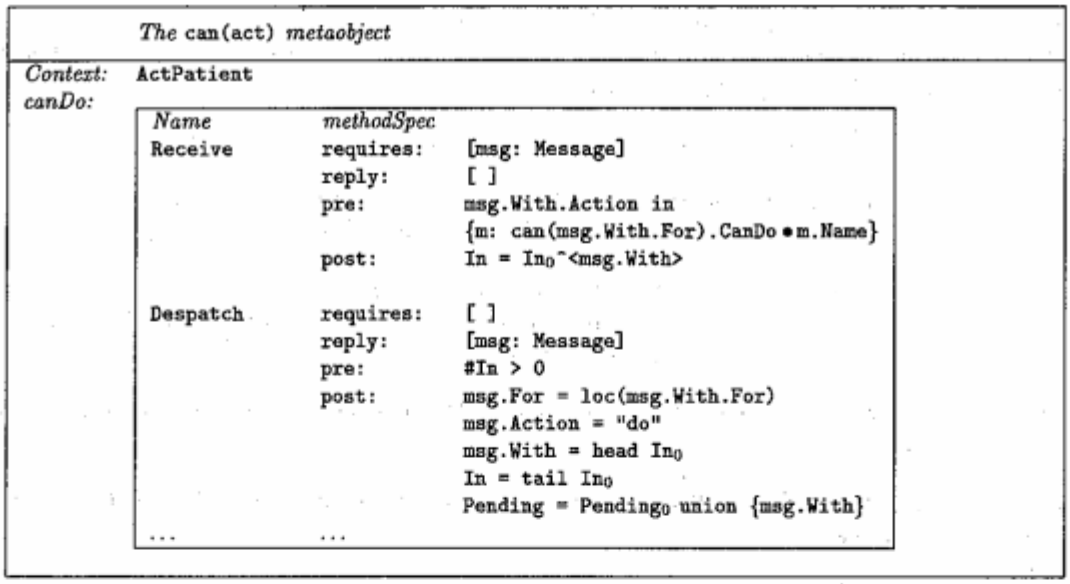

Figure 9: The can metaobject for an act metaobject

- `Despatch`: it may despatch a message, but only if there is one in the input queue. The precondition checks the size of the input queue using the set cardinality operator. `#In` represents the size of the queue. The message is sent to the `loc` metaobject of the object to which the message was originally sent. The action is to `"do"` the procedure associated with the message at the head of the input queue. The new input queue is formed from the tail of the existing queue.

Of course, the activity surrounding a given object could be *dramatically* affected by altering either of these methods, as just specified. We could change the `Receive` method so that it always placed certain messages at the head of the queue. For example, if there was a message `Emergency` that was to be expedited, we could alter the postconditions of `Receive` to be:

$$\texttt{msg.With.Action = "Emergency" => In = <msg.With>\^{}In}_0$$

In this way, we require that such a message be placed at the front of the queue. Alternatively, we could require that messages from certain objects always be placed at the head of the queue. We could alter the `Despatch` method so that it never despatched a message for an object if there was already one pending for that object. We could specify this by adding an additional constraint in the postconditions for that method:

$$\texttt{msg.With.For not in \{p:Pending}_0 \texttt{ • p.For\}}$$

The object involved in the despatched message does not appear in $\texttt{Pending}_0$ which is the before version of the set of messages currently being processed.

4 Cooperation through reflection

The R-OK model, through its metaobject mappings, provides access to the core of an information system, and to the behaviour of that core. This kernel includes such basic activities as (1) the instantiation

of an object, and (2) the assignment of a new value to some attribute of an existing object. Through this model, we have access to these activities *because they are embodied by the metaobjects* and so we are able to modify the expected behaviour. In this section, we will investigate two examples of how cooperation may be achieved through the reflective adjustment of the default behaviour.

4.1 Cooperative Instantiation

Nuclear Medicine is one of a family of related diagnostic services known as medical imaging that are available at most medium to large hospitals. It is an aid that is concerned with *physiology*, that is, the functioning of organs and bones. Other diagnostic aids, such as cat scans, ultrasound and X-rays are concerned with *anatomy*, that is, the shape or structure of the organs and bones. The most common types of diagnosis performed by a nuclear medicine department within a hospital are liver, bone, lung, cardiac and renal (kidney) scans.

Booking a scan: Typically, a general practitioner will refer a patient for a scan. That person will contact the department directly to arrange an appointment time. The receptionist will ask the person for information such as the type of scan, the patient's name, address, phone number, sex, date of birth, any drugs used or allergies, and the referring doctor.

Pre-injection: At this stage it is important that the technologist responsible for taking the scan have access to certain aspects of the patient's medical record or history. These will include whether or not the patient is pregnant, whether they are on medication, what allergies they may have, any related scans that they might have undergone, either within this department, or within others, such as X-rays or CAT scans.

Post-scan: It is quite common for a scan to raise further questions, ones that were not anticipated. For example, does the patient only have one kidney because of a car accident or because they were born with only one? Or, what is the norm for a person of this age and sex? This latter question may be answered by accessing a number of "Teaching Files" which have been built up by the hospital over a number of years.

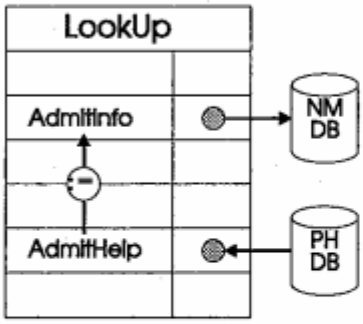

Figure 10: Getting `AdmitInfo`

The `state(Booking)` *metaobject*

Context: `Booking`
State:

Attribute	*Type*
[PatientId:	CharString;
ScanType:	CharString;
ScanDate:	Date;
ScanTime:	Time;
AdmitInfo:	[On:Date; AtDept:CharString];
XrayInfo:	[On:Date; Pic:Image];
CatInfo:	[On:Date; Pic:Image];
AdmitHelp:	[On:Date; AtDept:CharString];
XrayHelp:	[On:Date; Pic:Image];
CatHelp:	[On:Date; Pic:Image]]

Figure 11: The `state(Booking)` metaobject

Suppose patient 117895 is scheduled to have a lung scan on 27-Oct-1995 at 11:30am, and the Nuclear Medicine Department would like to have, by that time, as much relevant information as the hospital

can supply. This might include details of previous admissions, recent chest X-rays, and so on. We may imagine that a message of the form:

```
Help [id is 117895, scan is lung, on is 27-Oct-1995, at is 11:30:00]
```

is to be sent out. The message supplies the patient's Id, the type of scan he or she is to have, and when the scan is to take place. The overall intention is to broadcast the message as quickly and as widely as possible across the various database sites within the hospital. In general, whenever a booking is made, a request for cooperation is issued. Effectively, this involves the transmission of a message of the form:

> "What do you know about patient `PatientId` who is scheduled for a `ScanType` scan? We need to know by `Date` and `Time`."

The request for assistance and information is, essentially, a request to copy information from one site to another; see Figure 10. To effect this copying, we define a `Booking` object of the form shown in Figure 11.

The attributes of this object fall into two groups. The first group contains four attributes that are supplied directly from the booking process itself. The other group contains three pairs of attributes which arise as follows. Suppose that there are three remote databases that have the potential to help the department: (1) the Patient History Database will have information on the date on which the patient was first admitted to the hospital, and the department involved. (2) the X-ray Database might contain images of that patient, ones related to the type of scan the patient is to undergo; we would like the most recent of these, if any. (3) The CAT Database might also have relevant images. Each of these databases gives rise to two attributes. In the following discussion, we will only consider information on patient admissions. The attribute `AdmitHelp` is linked to the Patient History Database and will contain any information the database can supply. This information will be copied across to the `AdmitInfo` attribute which will be linked to the Nuclear Medicine Database. The instantiation of a booking object is, by itself, enough to trigger off requests for information.

The instantiation process is described by a `create` procedure that is stored in the `loc(Booking)` object that is shown in Figure 12. As discussed in subsection 2.4, when the interpreter encounters an unrecognised symbol, it uses the *LookUp* attribute as a symbol table. If the symbol appears on the right-hand side of an assignment statement then a `get` message is sent to the associated surrogate object. If the symbol appears on the left-hand side, then a `set` message is sent. Whilst executing the `create` procedure, the interpreter must not be blocked by any of the last three assignment statements. It must somehow pass to the next assignment as soon as it has fired off processing on the previous one. The || sign indicates this. Thus the three-part statement:

```
AdmitInfo:=AdmitHelp || XrayInfo:= XrayHelp || CatInfo:=CatHelp
```

triggers three parallel processes. These are monitored by (the interpreter, acting as a method of) the `loc(Booking)` object.

From Figure 13, we see that the cry for help, in the form of a request to instantiate a `Booking` object, is received by `act(Booking)`. This is passed on to `loc(Booking)` as a `"do Create"` request. The `create` procedure is executed, causing `get` messages to be sent to each of the three *Lookup* objects, shown as shaded circles.

These objects have knowledge of the physical siting of the three databases and will send remote procedure calls which are received by `act` objects defined at each of the sites. Each of these will then activate the corresponding `loc` objects which will perform the necessary retrieval(s).

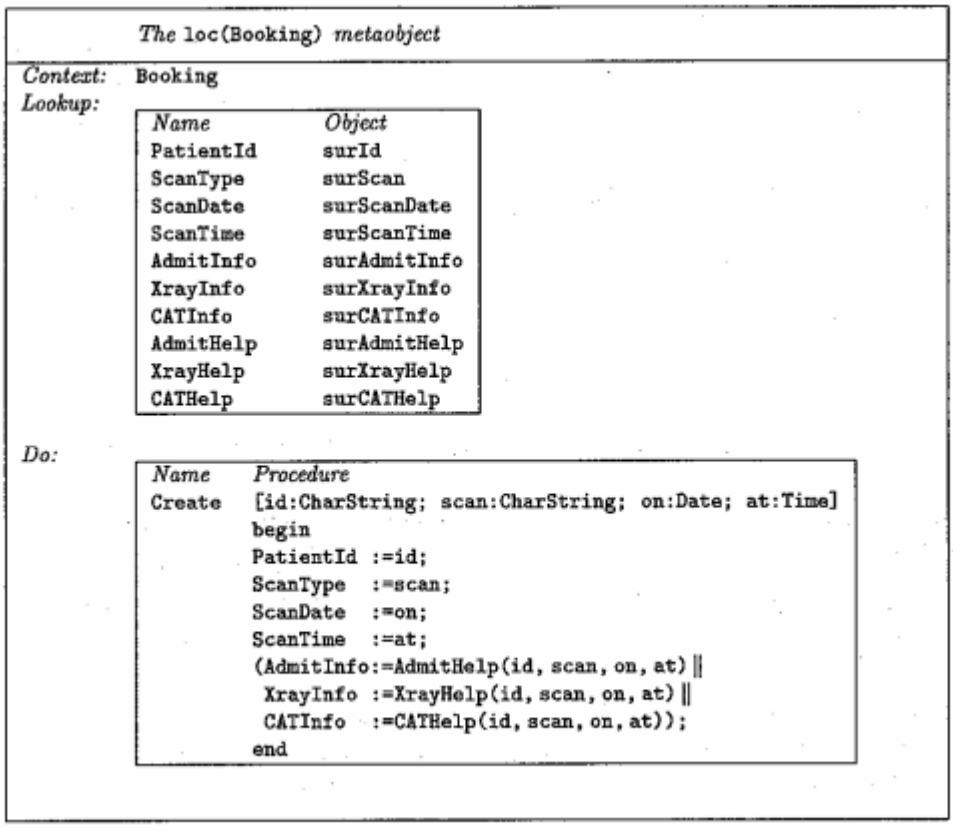

Figure 12: The loc(Booking) metaobject

4.2 Triggers

The second example of cooperation through reflection is one where the raising of the value of an attribute above a given threshold triggers an emergency action in some related object.

Suppose we have a patient object Patient with an attribute Temp that contains the patient's temperature. Suppose, also, that there are a number of activities which may or may not alter the value of this attribute, and it is difficult to tell, merely from the invocation of a patient method, that the temperature attribute is going to be changed. A more direct approach is to monitor activity involving the surrogate object associated with the Temp attribute. In the *Lookup* attribute of loc(Patient) there will be a pairing, say (Temp, surTemp), which redirects any actions involving Temp to the object surTemp. Such actions will typically involve gets and sets, as previously explained. We can monitor this activity by constructing an act(surTemp) object which will receive *every* action relating to patient temperature and which, in particular, can preview all set actions, testing the new value to see if it is above the threshold. If it is, then the act(surTemp) object can send a message to an Alarm object which is awaiting such notification. (An alternative is for act(surTemp) to notify Alarm every time the temperature changes and for Alarm to decide whether the threshold has been reached.)

The situation is a little more complex because we do not want Alarm to wait until it receives a signal before it invokes some Emergency action. Rather, we want a situation where the Emergency procedure is as advanced as it can be without taking precipitate and unwarranted action. We accomplish this in the following way:

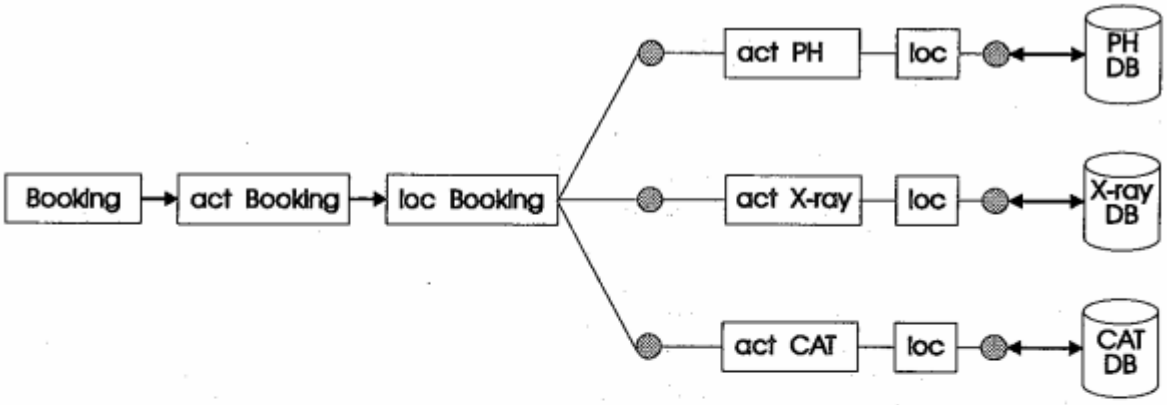

Figure 13: Cooperation through instantiation

- We introduce, into the `loc(Alarm)` object, the same `(Temp, surTemp)` pairing that was found in `loc(Patient)`.
- The `Emergency` procedure that activates the alarm devices includes a statement:

  ```
  wait until Temp > 40
  ```

  The interpreter, when it encounters such a statement, finds `Temp` in its *Lookup* table and sends a `get` message to `surTemp`. The value of `Temp` is returned and, if the value is not greater than 40, then the interpreter sends a `wakemewhen` message to `surTemp` and suspends itself.
- Henceforth, every `set` message sent to `surTemp` is examined and, if the new value is above 40, then a `resume` message is sent to `loc(Alarm)` which causes the interpreter to resume at the first statement after the `wait` command.

5 Summary

In this paper we have introduced the concept of using reflection to support cooperation processing activities between a number of internetworked autonomous database systems. Functional and operational reflection is manifested in a handful of special purpose meta-level objects which surround each individual database, turning hitherto conventional databases into a set of cooperating processes.

In particular, we introduced R-OK, a reflective model for distributed database systems, which employs a group of four special kinds of meta-objects to describe, synthesize and monitor the state and activities of objects originating from discrete database systems. The model, through its meta-object mappings, provides access to to the core functionality of a cooperative information system, and to the behaviour of that core. Through the R-OK model, one can gain access, and may tune, activities found in any modern distributed environment, such as instantiation of distributed objects and distributed method despatching.

References

[BB+91] Bartsch-Spörl B., Bredewig B. et al. (1992). Studies and Experiments with Reflective Problem Solvers, *ESPRIT Basic Research Project P3178 REFLECT*, Document RFL/BSR-UvA/II/2/1

[BR+90] Bartsch-Spörl B., Reinders M. et al. (1990). A Tentative Framework for Knowledge-level Reflection, *ESPRIT Basic Research Project P3178 REFLECT*, Document RFL/BSR-ECN/I.3/1

[Dav80] Davis R. (1980). Metarules: Reasoning about Control. *Artificial Intelligence*, 15, 179-222.

[EPT95] Edmond D., Papzoglou M. and Tari Z. (1995) "R-OK: A Reflective Model for Distributed Object Management", to appear in Procs of RIDE'95 (Research Issues in Data Engineering), Taiwan.

[vHa91] van Harmelen F. (1991). *Meta-level Inference Systems*, London, England: Pitman.

[HY88] Honda Y. and Yonezawa A.(1988). "Debugging Concurrent Systems Based on Object Groups", Procs of ECOOP'88: European Conference on Object-oriented Programming, Oslo, Norway.

[KRB91] Kiczales G., des Rivières J. and Bobrow D.G. (1991). *The Art of the Metaobject Protocol.* Cambridge, Mass., USA: The MIT Press.

[Mae87] Maes P. (1987). "Concepts and Experiments in Computational Reflection", OOPSLA'87.

[Mae88] Maes P. (1988). "Computational Reflection", *The Knowledge Engineering Review*, 3(1), 1988, 1-19.

[MM+92] Masuhara H., Matsuoka S. et al. (1992). "Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently", OOPSLA'92.

[Pae90] Paepcke A. (1990). "PCLOS: Stress Testing CLOS", OOPSLA'90.

[Rao91] Rao R. (1991). "Implementation Reflection in Silica", ECOOP'91, Lecture Notes in Computer Science 512, Springer-Verlag.

[SSF92] Stemple D., Sheard T. and Fegaras L. (1992). "Linguistic Reflection: A Bridge from Programming to Database Languages", Procs. of the 1992 Hawaii Conf. on Systems and Sciences, Koloa, Jan. 1992.'

[WSB92] Wielinga B., Schreiber A. and Breuker J. (1992). "KADS: a modelling approach to knowledge engineering" *Knowledge Acquisition*, 4, pp 5-53.

[Yo+91] Yokote Y. et al "Reflective Object Management in the Muse Operating System", Procs. 1991, Int'l Workshop on Object-Orientation in Operating Systems, also in Selected Technical Reports, Sony Computer Science Lab. Inc., May 1992.

[Yo92] Yokote Y. (1992). "The Apertos Reflective Operating System: The Concept and its Implementation", OOPSLA'92.