Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Designing Active Objects in DEGAS

J.F.P. van den Akker, A.P.J.M. Siebes

Information Systems (INS)

INS-R9702 February 28, 1997

Report INS-R9702
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Designing Active Objects in DEGAS

Johan van den Akker
Arno Siebes

CWI
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
e-mail: {vdakker,arno}@cwi.nl

ABSTRACT
This report discusses application design for active databases, in particular for the active object-based database programming language DEGAS. In DEGAS one modularisation principle, the object, is applied to all elements of the application, including rules. We discuss a design process consisting of four phases, corresponding with the four kinds of capabilities in a DEGAS object, attributes, methods, rules, lifecycles. The elements of this design process are similar to those found in a design methodology such as OMT. To illustrate the design process we use the example of workflow management. In addition, it shows that the application of one modularisation to all elements of an active database leads to a clear modularisation of the workflow application, Furthermore, this modularisation facilitates all important workflow evolutions.

1991 Computing Reviews Classification System: H.2.1 [Information Systems]: Logical Design - *data models*, H.2.8 [Information Systems]: Database Applications, H.4.1 [Information Systems]: Office Automation.
Keywords and Phrases: active databases, database design, workflow management.
Note: This project is supported by SION, the Foundation for Computer Science Research in the Netherlands through Project no. 612-323-424.

1 Introduction

Active databases [19] extend traditional databases with active rules, or triggers. From their original use for specific database tasks, such as constraint enforcement, the use of active rules has evolved to include large parts of information systems functionality, especially as an implementation mechanism for business rules [11]. As a consequence, the specification of rules must become an integral part of information system design.

Modularisation is generally accepted as a necessary tool for the design and understanding of computer software. Naturally, this also applies to rules in active databases. There are two approaches to the modularisation of rules. Either specific modularisation mechanisms are applied to the rules orthogonal to other modularisation, or one integral modularisation mechanism is applied to all elements of the database. The former approach is usually followed in systems with a separate rulebase, such as [18]. Here, the notion of rule sets is used to determine what rules are active and what not. This modularisation is also found in some object-oriented systems, such as SAMOS [10]. A drawback of this approach is the presence of an additional modularisation mechanism besides object-orientation.

In DEGAS [3], a temporal, active object database developed at CWI, we use the latter approach for modularisation of rules. Like other elements of the database, rules are encapsulated in objects. This encapsulation is motivated by the observation that rules define object behaviour. Thus, there is no separate rulebase and no need for additional modularisation concepts for rules.

Research on active database design has mainly focussed on analysing rule sets in order to check desirable properties, such as termination and confluence. In this area modularisation has also been addressed, for example in [6]. This work, however, focussed on partitioning a given rule set. Relatively little attention has been paid to the question, how we get a rule set for a certain application in the first place. In this paper, we investigate this issue. In particular, we look at the elements of an object-oriented design method, such as OMT [16], that are of importance to the derivation of rules in an application.

As an example in this paper, we use workflow. The work of many organisations is centered around workflows. A workflow is an activity involving the coordinated execution of multiple tasks performed by different processing entities [17]. Classical examples of workflow are the processing of insurance claims and of loan requests. Current implementations of workflow are mostly in separate workflow management systems (WFMS). Since most workflow management involves a lot of data, a lot of interactions between WFMS and database take place. At the same time, the Event-Condition-Action rules of active databases add to a database management system the kind of reactive capabilities also found in a WFMS. Hence, we expect that in the future workflow management will be merged into active database management systems.

Previous work on workflows in active databases is reported in [7] and [12]. In general, the use of an active database improves the data handling capabilities in the workflow. This pertains to the application data, as well as to the workflow management data. Hence, approaches to workflow based on active databases such as [7] provide models of the data involved in the workflow. This is in contrast with work such as reported in [1], based on Petri nets, with an inherent focus on the specification of the dynamics of a workflow.

A drawback of the approach in [7] is the lack of modularisation in the rulebase. A large set of rules is generated for a workflow, which is only partitioned afterwards for analysis purposes. Hence, a separate modularisation is applied to the rules. Furthermore, this modularisation is not used in the design phase. In this paper, we consider the design of a workflow with the other of the two approaches to rule modularisation mentioned above. In addition, we make the modularisation during the design of the application, using design principles formulated in this paper. We show that DEGAS allows us to modularise workflow management in a way that separates concerns and that promotes flexibility. In particular, it offers a framework to implement the workflow evolution policies described in [8].

Roadmap In Section 2, we briefly introduce the DEGAS model. Then, we state the DEGAS database design principles in Section 3. The next section, Section 4, discusses the specification of a workflow. In Section 5, we apply the guidelines to develop a design for workflow enactment. The following section, Section 6, shows that this design offers the necessary flexibility for evolution of the workflow. The last section contains conclusions and directions for future research.

2 A short intro of DEGAS

We now give a concise introduction to the main concepts of the DEGAS data model. It is based on autonomous objects. The motivation for object autonomy is on one hand a natural further development of active object-oriented databases and on the other hand the development of highly distributed information systems. The main contributions of DEGAS are:

- The integration of historical and active database functionality.
- A straightforward mechanism for object evolution, especially suited for implementing roles.
- Complete encapsulation of an object's behaviour, including rules.
- A good formalisation of rule semantics.
- A conceptual model for distributed information systems.

For a more elaborate introduction of DEGAS the reader is referred to [3]. A full formal definition of DEGAS can be found in [2].

The fundamental notion in DEGAS is the object. The definition of an object in DEGAS consists of structure and behaviour. The structure of an object is defined by its attributes. The behaviour definition of a DEGAS object consists of three elements: methods, lifecycles and rules. Methods define the actions an object can execute. The lifecycle of an object specifies sequencing and preconditions of methods. A rule states that an object will execute a given action in certain situations, specified by events and conditions on object states.

In other words, methods and lifecycles define the *potential* behaviour of an object, whereas rules describe its *actual* behaviour as far as can be pre-determined. Conventionally, only potential behaviour is specified in an object.

Figure 1 shows an example DEGAS object modelling a PIN card. Attribute and method specification is straightforward in DEGAS. Lifecycles are guarded basic process algebraic expressions [5] composed from the set of method names as basic actions using the sequential composition (;), alternative composition (+), repetition (*), and parallel merge (||) operators. For example, the third line of the lifecycle definition in our example specifies that a *ReqWithdraw* action must be followed by a *WithdrawOK* or a *WithdrawRefuse* action, and that this sequence may be repeated arbitrarily. The parallel merge operator || means that two actions take place without restriction on their sequence, i.e., $A \| B = A;B + B;A$.

Rules in DEGAS follow the usual Event-Condition-Action (ECA) format. The informal semantics of an ECA rule is, that if the event occurs and the object satisfies the condition, the action is performed. In DEGAS events are specified the same as lifecycles with addition of a negation operator (¬). Conditions in lifecycles and rules can refer to historical values of attributes. If an attribute name is parameterised by a timestamp, it refers to the value of the attribute at the specified time. Otherwise, it refers to the current value of the attribute. The rules in PINcard show historical references in DEGAS rules.

More in particular, the first rule specifies that the PINcard sends its permission for a cash withdrawal after a request, if the total amount withdrawn during the preceding week is less than the limit of the card. The second rule responds with a refusal, if the limit is exceeded.

The class of a DEGAS object specifies its inherent *capabilities* (= attributes, methods, lifecycles and constraints). Object specialisation in DEGAS is achieved through addons. An addon models transient capabilities of an object. Addons can be added to and deleted from an object dynamically, for example, when an object engages in a relation. A restricted form of inheritance is supported by DEGAS. Since this is not relevant for this paper, the interested reader is referred to [2] for more details. Relations in DEGAS are also objects with structure and behaviour.

Relations between objects are objectified in DEGAS, as is often the case in information systems modelling methods such as NIAM [15]. The motivation for this is a formal view of a relation as a kind of contract. Moreover, it ensures that relations can engage in relations themselves.

Before two objects enter a relationship, certain preconditions will have to be satisfied. For example, if two persons wish to marry, both must be of a different sex and unmarried. Likewise, the termination of a relationship is subject to restrictions. In a lot of relations we need to store data and behaviour of the relation. An obvious example is the bank account relation between a bank and its clients. This information and the capabilities to handle termination of the relationship are stored in a relation object. The capabilities to handle the initiation of a relationship, including the creation of the relation object, can be found in the corresponding relation class object. A relation is initiated by sending an `initiate` message to the relation class object. Depending on the application semantics of the relation the relation class object finds a matching partner and checks the preconditions of the relation.

An object that engages in a relation is extended using the addon mechanism, because it must gain capabilities to deal with the relationship. For example, once a person has a bank account, he can transfer money to other bank accounts or withdraw money through a cash dispenser. Any kind of capability can be defined in an addon. An addon, however, cannot exist independently, like an object or a relation object. An object is extended with an addon by executing an `extend` action with the addon's name. Likewise, an addon is removed by a `delete` action.

```
Object PINcard
Attributes
    number : integer
    limit : integer
    account : Oid
    issuer : Oid
    owner : Oid
    PIN : integer
Methods
    ReqWithdraw(amount:integer,requester:Oid) = {

    }
    WithdrawOK(amount:integer,requester:Oid) = {
        requester.allowed(amount)
    }
    WitdrawRefuse(amount:integer,requester:Oid) = {
        requester.refuse(amount)
    }
    ChangeLimit(newLimit : integer) = {
        limit = newLimit
    }
    ChangePIN(newPIN : integer) = {
        PIN = newPIN
    }
Lifecycles
    ([sender==issuer] ChangeLimit)*
    ([sender==owner] ChangePIN)*
    (ReqWithdraw;(WithdrawOK + WithdrawRefuse))*
Rules
    On (WithdrawOK(amount,atm)(t))*;
        ReqWithdraw(reqAmount,machine)(t1)
        if t1 − Min(t) ≤ 1 week
            && Sum(amount, t)+reqAmount ≤ limit
        do WithdrawOK(reqAmount,machine)
    On (WithdrawOK(amount,atm)(t))*;
        ReqWithdraw(reqAmount,machine)(t1)
        if t1 − Min(t) ≤ 1 week
            && Sum(amount, t)+reqAmount > limit
        do WithdrawRefuse(reqAmount,machine)
EndObject
```

Figure 1: A DEGAS object

3 Design Guidelines for DEGAS

In modelling applications for DEGAS, we need a number of guidelines. Since the central notion of DEGAS is object autonomy, we first recapitulate the criteria for object autonomy given in [2]:

- Complete encapsulation of the behaviour of an object.
- Strictly regulated access to an object.
- Minimal guarantees about an object's behaviour towards other objects.
- Minimal dependency of an object on the behaviour of other object.

These criteria were used to guide the development of the DEGAS data model. Naturally, they have their consequences on DEGAS database design. In one sentence, we can say that DEGAS objects combine minimal capabilities with maximal encapsulation.

Minimal capabilities means, that at any time an object only possesses the capabilities it needs. This applies to both time and place of information storage. An object gets information only when it needs it. Likewise, if it needs information from another object, it will request that information when needed. An application of this principle is, that an object is extended with extra capabilities, when it enters a relation. If it is not in a relation, the object does not have the information associated with that relation.

Maximal encapsulation means, that everything is defined on the object itself. It is one of the main consequences of object autonomy. Every aspect of the behaviour of an object is defined on the object itself, including rules. The modularisation primitives for the rules are the notions of object-orientation, that are also applied to attributes, methods, and lifecycles.

In database design terms, the guiding principle can be rephrased as follows: An object gets only the information it needs, but it does get all the information it needs. This has the additional advantage, that NULL-values have only one meaning in DEGAS: It means that the value of the attribute is unknown. It does not mean that the attribute is not defined, since attributes only exist during the time they are needed.

These design principles are applied to a database design process. Design in DEGAS encompasses four dependent phases:

1. Identify objects in the application and the information they possess.
2. The actions of an object
3. The activation of each action (with static constraints)
4. The lifecycle of an object (i.e. dynamic constraints)

Phase 1 The first phase of database design in DEGAS is concerned with the static part. It consists of identifying the objects and their relations, and determining the information contained in them. The identification of objects and their relations in a DEGAS database design is not radically different from usual object-oriented design techniques [16]. Next, we determine the data in each object. Here, we have to make a distinction between the data that is always present, and the data that is dependent on the presence of a relation with another object. The former are called *inherent* attributes, the latter are *transient* attributes. Suppose we have a `person` object, then inherent attributes are things like name, birth date, birthplace, and sex.

The part of an object associated with a certain relation is generally called its *role* in the relation. The concept of roles in an object-oriented context is elaborately discussed in [20]. The capabilities an object has to deal with a certain relation are modelled by a role. Since these capabilities are only needed when the object is involved in the relation, these are called *transient* in contrast with the object's permanently present inherent capabilities. In DEGAS transient capabilities of objects are defined in addons. Hence, all information associated with a role is implemented by an addon.

A guideline in determining relations between objects is given by their information exchanges. If an object gets information from another object, it must have a relation with it. The other way round is also true: An object can only communicate through its relations. Hence, we use the information flow in an application to determine the relations in the application domain.

The result of this design phase is a static DEGAS object model. We have defined the objects, their relation and associated addons. Furthermore, we have defined the attributes in each of these. Each following phase in DEGAS database design are all executed iteratively. First, the inherent capabilities of objects are addressed. Then, we specify the capabilities of addons.

This iteration originates in the semantics of DEGAS addons. In the specification of an addon, we can use those capabilities of the object being extended, that we are certain to be present. In other words, an addon can use the capabilities inherent to the object it extends, and the addons it assumes present, as declared in the **extends** specification. If we assume a directed edge from each relation to its partner objects, then a DEGAS database design, is a Directed Acyclic Graph (DAG) with the objects as the leaves, as is depicted in Figure 2. Hence, we can start out from the leaves in the DAG and then progress towards the highest level relations.

Phase 2 After the specification of the information, we look at the dynamics of the objects. This means that we have to identify the actions that can be executed on the information in the objects. In addition to this, engaging and disengaging in relations are also actions. In the resulting DEGAS database, these actions are the methods of the objects.

Initially, the approach to finding the methods is a "shopping list" approach, as it is called by Meyer [14]. The actions of an object can be either services to the outside world, or internal state transitions. In phase 2, this distinction is not of importance.

This phase gives us information to check the result of the previous phase. For each of the actions, we can determine the information it needs. This information must be either available in the object itself, in the form of an attribute, or it is obtained through a relation. Hence, we

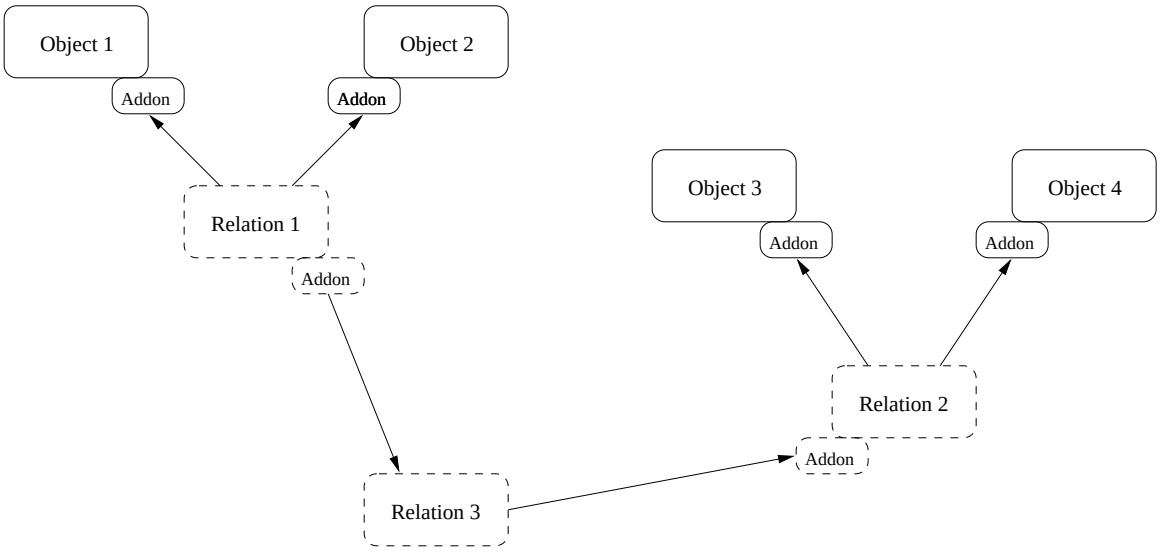


Figure 2: A DEGAS database design as a Directed Acyclic Graph

can check whether we have specified all attributes of an object. Furthermore, we can check whether there is a relation for every information exchange between objects. Vice versa, we can check whether every relation is used to exchange information between objects.

Phase 3 Having specified the actions an object can execute, we have to specify when these actions are executed. We do this by specifying the situations that trigger the actions. The specification formalism of these situations is up to the designer. He can use a graphical formalism, such as the situation diagrams introduced in [13], or another formalism. The only requirement is, that it has a clear translation to ECA rules.

In this phase, we first specify the activation conditions inherent in an object, i.e. in an object without any addons. After that, we specify the interaction scenario for each relation. This interaction scenario describes the communication between the partners in the relation and the relation object. From this scenario, we derive the activation conditions for the actions of the objects. Please note, that these interaction scenarios can involve inherent actions of a partner object, since these are available to addons.

Activation conditions thus derived can be either local to an object or the result of actions of another object. The first category is found back in rules in the object itself. The second category means invocation of a method from another object, either from a rule or from a method of that object.

For some methods we are not be able to specify an activation condition within the application. This is the case for activations either by users or by other software components. In these cases, we also specify an interaction scenario for the interface relation to these agents. This interaction scenario specifies what actions can be invoked by a user or another piece of software.

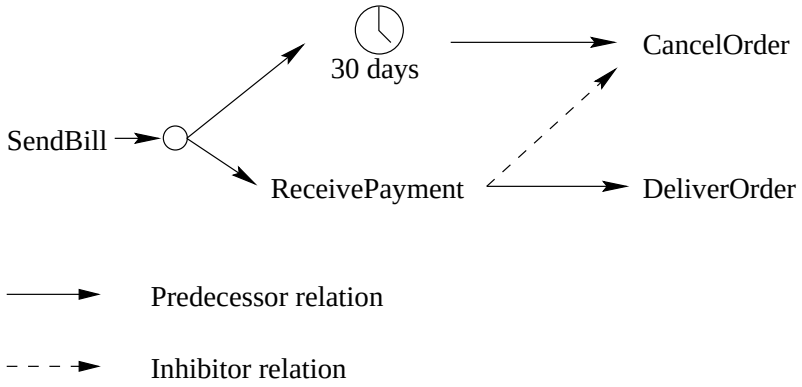


Figure 3: Part of an order processing workflow with a clock and task inhibition

Using the result of this phase, we can validate the list of actions specified in the previous phase. If we are not able to formulate an activation condition for an action, it is most probably not needed in the current application.

Phase 4 The last phase in the design of a DEGAS database, is to specify the temporal relations between the actions. These are meant to express the ways an object can execute actions. The result of this phase is the lifecycle of an object. The lifecycle specified for an addon conforms to the lifecycle of the inherent object by the use of the communication merge to merge the lifecycles of objects and addons. For a more elaborate discussion of lifecycle merging the reader is referred to [4]. In terms of OMT, the lifecycle gives the dynamic model of the application, without the activations of the state transitions.

The lifecycle provides a check on the activation conditions of the previous phase. If the activation condition contains an event expression, then this event expression must comply to the lifecycle specified in Phase 4. A conflict here means that either the activation or the lifecycle is incorrect.

4 Specification of workflow

The example we use in this paper to show the DEGAS design process is workflow management. In the specification, we are mainly concerned with routing of an activity. For each processing phase, we store the immediately preceding and succeeding tasks. This specification of the routing of a workflow allows us to specify the conditions to start each processing phase in the workflow. These conditions depend on the way it is related to its predecessors. The different types of relations between tasks are called *routing elements*.

The set of preceding tasks of a task τ is denoted by the set $pred(\tau)$. Likewise, the set of succeeding tasks is denoted by $succ(\tau)$. As an example consider the workflow in Figure 3, which gives the workflow for billing an order. A bill is sent to the customer. If we do not receive payment in 30 days the order is cancelled. Otherwise the order is delivered.

In this workflow we have the following predecessor and successor tasks:

$succ$(sendBill) = {timer(30), ReceivePayment}	$pred$(sendBill) = $\emptyset$
$succ$(timer(30)) = {CancelOrder}	$pred$(timer(30)) = {sendBill)}
$succ$(ReceivePayment) = {DeliverOrder}	$pred$(ReceivePayment) = {sendBill}
$succ$(CancelOrder) = $\emptyset$	$pred$(CancelOrder) = {timer(30)}
$succ$(DeliverOrder) = $\emptyset$	$pred(\zeta)$ = {ReceivePayment}

Besides positive predecessors, we need the notion of negative predecessors, or *inhibitors* for a task. These are tasks, that prevent the execution of another task. In the above example, we cancel an order of a customer who has not paid his bill in 30 days. Clearly, this task is inhibited by the payment of the bill. Hence, associated with each task τ is a set of inhibiting tasks denoted by $Inhib(\tau)$. In the example in Figure 3 we have:

$Inhib$(CancelOrder) = {ReceivePayment}

Other tasks have an empty set of inhibiting tasks.

The example in Figure 3 also contains a special kind of task, that is a timer. A timer is simply a task that is completed at the specified time past its start.. In this example, the effect is that the ReceivePayment only inhibits the cancellation of an order if it is completed before 30 days are over.

Furthermore, there might be certain conditions associated with the execution of a task on a job. One of the main uses of conditions is as a criterion to choose between a number of successor. For example, the billing procedure of a mail order company might make a difference between new customers and known customers. Hence, each task τ has an associated precondition $Precond(\tau)$. If τ has no specific precondition, $precond(\tau) = True$.

The Workflow Management Coalition distinguishes five routing elements [21], apart from simple sequential routing. These are AND-split, AND-join, OR-split, OR-join, and iteration. These routing elements can be formalised in terms of predecessor and successor tasks, and preconditions. This formalisation is translated to active rules to start each succeeding task. The formalisation and active rules for each routing element is found in the appendix.

5 Designing a workflow in DEGAS

In this section, we apply the design guidelines from the previous section to the example of workflow management. The minimal information and maximal encapsulation principle leads to a modularised approach to workflow management in active databases. Current approaches are all more or less global. In [7], groups of rules are only introduced after the rules have been derived. Although these groups are helpful in analysis of the rulebase, they are of no help during the design of the database. We show that the DEGAS approach to active database design leads to a clean database design.

In the following discussion, we will first consider workflow *in abstracto*. Then, this discussion will be illustrated by a concrete example, which is an order processing flow. The billing task specified in the previous section is part of this order processing. This flow is depicted in Figure 4

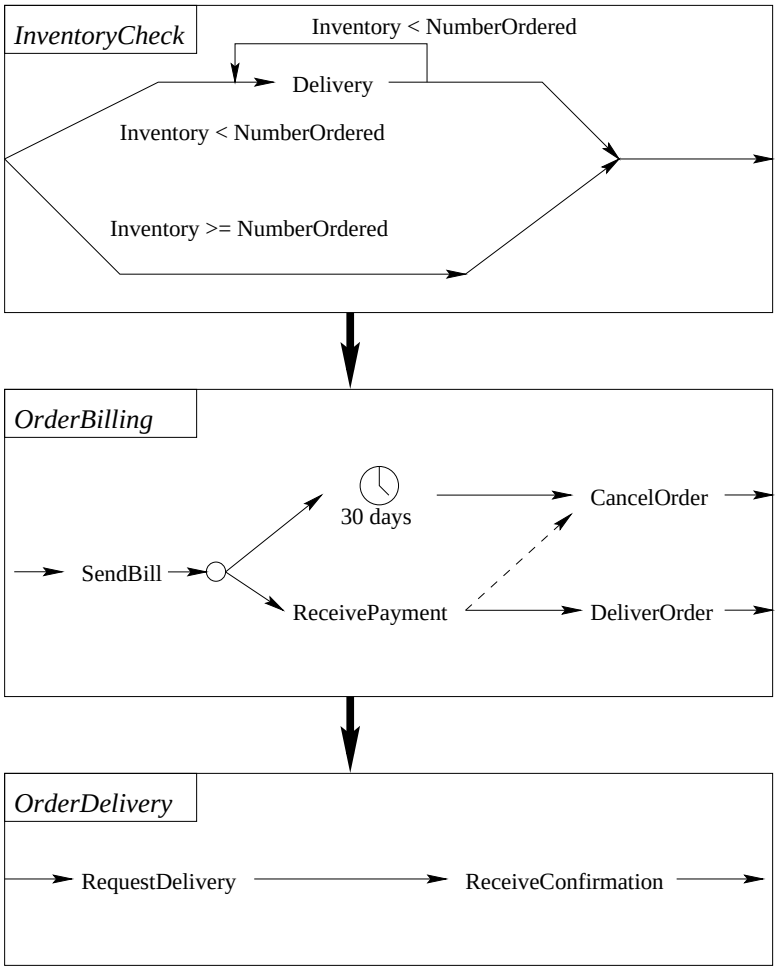


Figure 4: The workflow for order processing

5.1 Phase 1: Identifying the Objects

A workflow management system is there to support a *job* getting done. This job follows a certain activity or *schema*. This *schema* defines how the job is processed by the system. It consists of a number of processing phases, or *tasks* that must be executed in a certain sequence. These tasks are executed by *agents*. An agent can be a person or a computer program.

The `job` object contains the application data. In traditional, physical workflows this would be a form. For example, in our order processing workflow it is the software equivalent of an order form. Hence, it contains attributes like the item on order, the quantity and the negotiated price. Hence, its attribute specification is the following:

Attributes
item : string
number : integer
price : real
currentTask : number

The `agent` objects[1] implement the application functionality of the workflow. This means that they can represent anything from a piece of software processing the job to an interface to a person. Thus, `agent` objects form the interface to the outside world. in our example, on of the agents is the Inventory Controller, which keeps the number of items in stock and the reserved part of the stock.

Attributes
Inventory : integer
Reserved : integer

The third important piece of information in a workflow is routing information, embodied by schemata. A workflow schema describes the way a certain activity is completed. Such an activity is composed of a number of tasks that need to be executed in a certain sequence. Thus, the `schema` class is an additional class in our design. These objects store workflow schemata in terms of successors and predecessors to each task.

The relations between these three classes of objects are mainly determined by their information exchange. There are three pairs of object classes in our example, which means three potential binary relations. We briefly consider these three pairs. During the execution of a task of a job by an agent, the `agent` object needs information from the `job` object. Hence, a `job` object has a relation with the `agent` object that executes its current task. We call this relation the `TaskExecution` relation. The role of the `agent` is that of `processor`, while the `job` is `processed`. As a consequence of the minimality principle, explained in the previous section, this relation is only present while the agent executes the task. Once the task is completed, the relation is deleted again. In our example, the `InventoryCheck` relation is an example of a `TaskExecution` relation. Its only attributes are the partners in the relation, `Job` and `Schema`, that are specified in the **relation** clause.

In our order processing example, each phase has a separate `TaskExecution` relation. These are the `InventoryCheck`, `OrderBilling` and `OrderDelivery` relations.

[1] Not to be mistaken for any kind of *intelligent agents*.

The next pair is `job` and `schema` objects. Every job is routed according to some schema. Therefore, a job must be provided with routing information by a schema. This leads to a relation between a `job` and a `schema` object, named the `JobFlow` relation. In this relation, the `schema` has the role of `router`. The `job` is `routedBy` the relation. The `job` engages in this relation, as soon as it is started. Again the `JobFlow` relation object does not store any information.

The remaining pair, `schema` and `agent` do not have a meaningful information exchange. The `schema` object contains information about the way `job`s can be routed. This information is not necessary for an agent.

The result of this design phase in terms of generic workflow objects is depicted in Figure 5. Our concrete example is shown in Figure 6. In these pictures, the large rounded rectangles are objects. The small rounded rectangles denote addons. A dashed border indicates a relation object. Please note that the arrows do not imply any arity constraints on the relations. Instead, they point to the partner objects, on which the relation object depends for its existence.

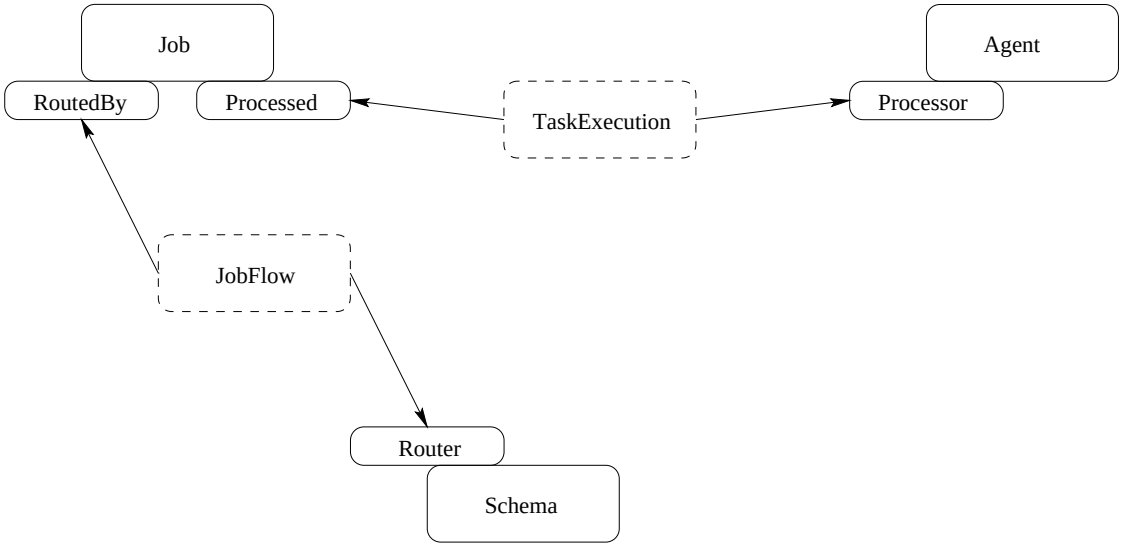


Figure 5: DEGAS object model of a workflow

5.2 Phase 2: The actions in a workflow

The next phase in the design of a workflow in DEGAS is to specify the actions of the different objects.

5.2.1 Inherent actions of objects

Each `agent` has actions to start its task and to signal the completion of its task. Further actions of an `agent` are dependent on the kind of agent. For example, the `InventoryControl` agent has actions to reserve stock for an order and to put newly arrived stock in the inventory.

Methods

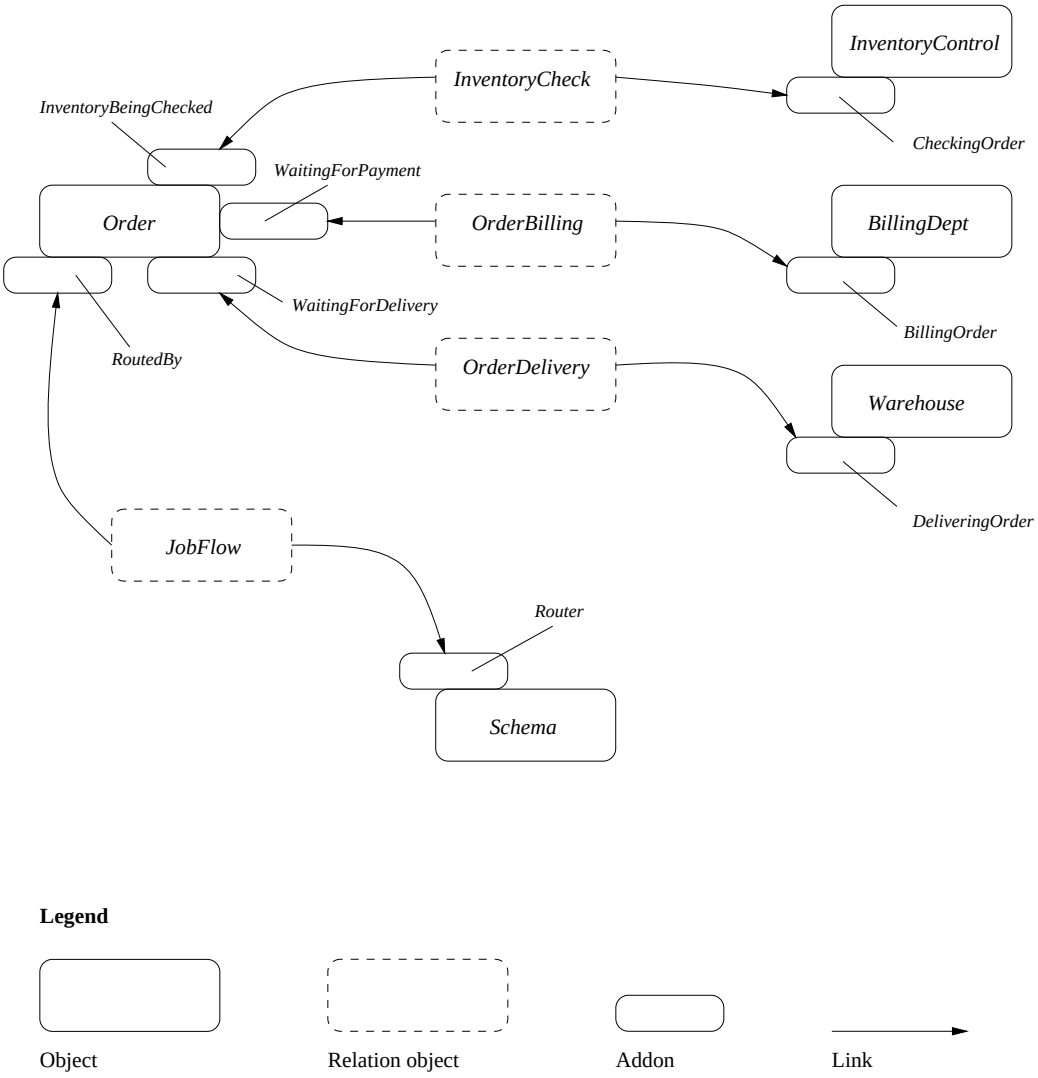


Figure 6: The object schema for an order processing workflow

```
reserve(number:integer) = {
    Reserved = Reserved + number
}
newstuff(number:integer) = {
    Inventory = Inventory + number
}
```

The only inherent action of a `job` object is the action to execute a certain schema. This action means that the `job` enters a `JobFlow` relation with a `Schema` object. Other actions may be defined for other purposes, but these are not relevant in this example. Since the **Extend** action is built-in in every DEGAS object, we do not see it back in the specification of the `order` object.

The services of a `schema` object are to provide information about the flow it defines. This means that it can answer the question, what comes after a specific task. The answer is provided by way of an addon that implements the routing decision to be made after each task. Hence, the only information a job needs to provide to get an answer is the job it has just finished. This is a consequence of the minimality of information principle. For example, in the workflow shown in Figure 6, if a job has finished the `InventoryCheck`, it requests the next task from the `JobFlow` relation object. It forwards the request to the `Schema` object, that replies with the name of the addon implementing the `OrderBilling` phase.

5.2.2 Transient actions of objects

Having defined the actions in the objects, we proceed to specify the actions in relations and their associated addons. With regard to a relation the most important issue is, what a relation enables the partners to do. From this, we can derive the actions of the relation itself.

The `JobFlow` relation enables the `job` to follow certain workflow. Through `JobFlow` it requests information on what task to execute next. Hence, the `RoutedBy` addon forwards a request for the next processing phase to the `JobFlow` relation object.

```
Addon RoutedBy
extends Order
...
Methods
    nextPhase(current:string) = {
        jobflow.whatNext(current)
    }
```

The `JobFlow` relation object contains an action to inform the `job` partner of its next task. The answer is given by instructing the `job` object to extend itself with an addon that contains the actions of the next task. A further action is to replace the `Schema` object in the relation with a new object.

```
Object JobFlow
...
Methods
    whatNext(task : string) = {
        succ = Schema.successor(task)
        Job.Extend(succ)
```

```
    }
    replaceSchema(newSchema:oid) = {
        Schema.delete(Router)
        Schema = newSchema
        Schema.extend(Router)
    }
```

The `Router` addon provides access to the routing information in the `Schema` object.

```
Addon Router
extends Schema
...
Methods
    successor(task:string) = {
        return succ(task)
    }
```

The `TaskExecution` relation allows a `job` to be processed by an `agent`. The existence of the relation means that the task will be executed, but it is not started immediately after the creation of the relation. Hence, the `Processed` addon will contain the necessary actions to start the task. In addition, it will contain actions that give the `agent` the information necessary to execute its task. The `Processor` addon will contain the actions, that are specific to this task. Hence, it can be regarded as an interface to the inherent actions of the `agent`.

In our order processing example, one specialisation of the `TaskExecution` relation is the `InventoryCheck`. This implements the first processing phase which is done by the `InventoryControl` agent. The `InventoryCheck` relation object implements actions to start and to finish the processing phase.

```
Object InventoryCheck
Relation InventoryControl, Order
...
Methods
    Start(number : integer) = {
        InventoryControl.request(number)
    }
    Finish() = {
        Order.EndInventoryCheck
    }
```

An `order` object can only be a partner in the `InventoryCheck` relation, if it is routed by some workflow schema. Hence, the `InventoryBeingChecked` addon extends an `order` object, that is already extended by a `RoutedBy` addon. This is specified in the **Extends** clause of the addon specification. The only action in this addon contains the functionality to end the phase.

```
Addon InventoryBeingChecked
extends RoutedBy
...
Methods
    EndInventoryCheck = {
        nextPhase
    }
```

Since the `InventoryControl` object is the agent for the `InventoryCheck` task, the `CheckingOrder` addon contains most functionality. This consists of actions to request the number of items required for the order and to reserve the items, if they can be supplied.

Addon CheckingOrder
extends InventoryControl
...
Methods
request(number : integer, dest : oid) = {
noOfItems = number
}
getit(number : integer, dest : oid) = {
reserve(number)
}

5.3 Phase 3: Activation of action

In the third phase of the DEGAS database design, we specify when the different actions of an object are executed. This means that we first specify the internal activations in inherent objects, which are derived from the static constraints of the objects. Then, we formulate the interaction scenario for each relation, which leads to activation of actions in the partners of the relation, as well as in the relation object itself.

The objects, defining inherent capabilities, have no relations with other objects. Hence, inherently no object has business rules. For our workflow application, there are no integrity constraints on the objects, `job`, `agent`, and `schema`. The inherent actions may be used, however, in the addons that extend objects.

The interaction scenario for the `JobFlow` relation is depicted in Figure 7. The `job` object, in our example the `order` object, requests the next task on completion of a task. The `whatNext` action of the `JobFlow` object gets this information by a call to the `successor` function of the `Schema` object. The `JobFlow` object then sends a message back to the `order` object. This is an `extend` action to add the addon implementing the next task, in this case the `OrderBilling` task.

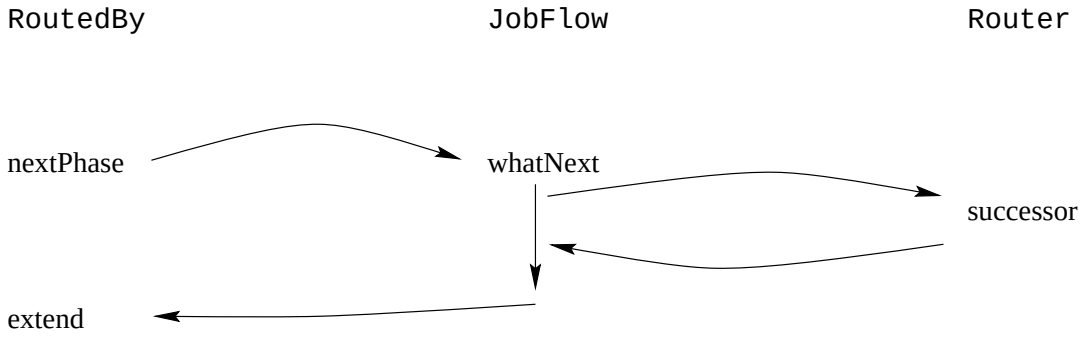


Figure 7: The interaction scenario for the `JobFlow` relation

The interaction scenario for the InventoryCheck relation is shown in Figure 8. The goal of this task is to check if the inventory suffices to deliver the order. The InventoryBeingChecked addon invokes the start action of the InventoryCheck object. At its turn it sends a request message to the InventoryControl object. When the inventory check is successful, the getit action is executed by the InventoryControl object. This action invokes the Finish action of the InventoryCheck, which leads to execution of the EndInventoryCheck action of the order object.

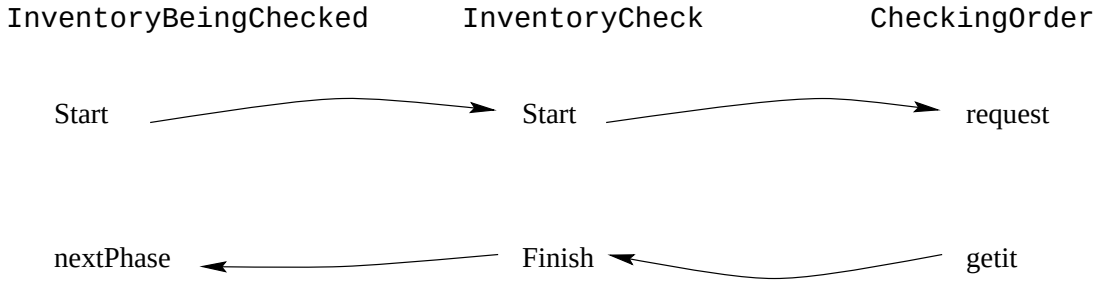


Figure 8: The interaction scenario for the InventoryCheck relation

The internal processing of the InventoryControl object is an iteration of the request action. As is shown in the workflow, the required number is requested from the stock. If this number cannot be reserved, the request is repeated each time new inventory arrives.

This leads to the activations of the different actions shown in Table 1.

5.4 Phase 4: Constraints on actions

The final phase of the design process is the specification of the order of the actions. This leads to a lifecycle for each object, that embodies its dynamic constraints. Lifecycles are specified in DEGAS using a process algebraic formalism [5]. Please note that the lifecycle of an addon can include inherent actions of the object it extends.

The order object does not have any inherent actions, so its lifecycle is empty. With regard to the JobFlow relation, it can enter one of these relations. During the existence of the relation, the order object can go to a next phase an arbitrary number of times. Hence, the lifecycle of the RoutedBy addon is:

Extend(RoutedBy);nextPhase*;Delete(RoutedBy)

The actions of an order object in the InventoryCheck relation are again limited. It enters the relation and executes one action to end the relation. Hence, the lifecycle of the InventoryBeingChecked addon becomes:

Extend(InventoryBeingChecked);
EndInventoryCheck;Delete(InventoryBeingChecked)

The inherent actions of the InventoryControl object can be executed in any desired order. Hence, the lifecycle is:

reserve*
newstuff*

Object or addon specification		
	Action	Activation
`order`		
	No actions specified	
`RoutedBy`		
	nextPhase	invoked by `EndInventoryCheck`
`InventoryBeingChecked`		
	EndInventoryCheck	invoked by `InventoryCheck.Finish`
`InventoryControl`		
	reserve	invoked by `getit`
	newstuff	invoked from outside scope of example
`CheckingOrder`		
	request	- invoked by `InventoryCheck.Start` - invoked by occurrence of `newstuff`
	getit	executed when a `request` occurs and the inventory is sufficient.
`InventoryCheck`		
	Start	executed on `order.extend(InventoryBeingChecked)`
	Finish	executed on `CheckingOrder.getit`
`JobFlow`		
	whatNext	invoked by `RoutedBy.nextPhase`
	ReplaceSchema	invoked from outside scope of example
`Router`		
	successor	invoked by `JobFLow.whatNext`

Table 1: Activations of actions in the workflow example

The order of the actions that pertain to the `InventoryCheck` relation reflects the scenario of the relation. A `request` action can be executed a number of times before a `getit` action is executed. Furthermore, the `request` call is only accepted from the object itself and the `InventoryCheck` relation object. The lifecycle of the `CheckingOrder` addon is:

```
request*;getit
([sender==InventoryCheck]request)*
([sender==self]request)*
```

The other relation in our example is the `JobFlow` relation. The relation object only has two actions, `whatNext` and `replaceSchema`. These can be executed in any order, but the `whatNext` action is only accepted from the `job` object, in our example the `order` object:

```
(sender==Job)whatNext*
replaceSchema*
```

The `schema` side of this relation is very simple. The only action is the `successor` action, that can be executed any number of times:

```
successor*
```

Complete Application This completes the design of our workflow example. The complete DEGAS source, that results from this design is given in the appendix.

6 Flexibility of the workflow

One of the chief characteristics to judge a workflow implementation is its flexibility. In particular, it must be easy to change elements of the workflow. This can be either the routing of a workflow, or the way a task is executed. In this section, we show that the workflow design of the previous section, using the DEGAS minimality principle, provides the necessary flexibility.

6.1 Evolution of the Workflow Schema

Over time, the schema of a workflow may evolve. The causes of workflow evolution can be very diverse. They might be optimisations due to an analysis of the process, new legal requirements on a production process, etc. In this subsection, we look at the effects of workflow evolution on the workflow schema, i.e. the changes it causes in the routing of the workflow. Changes can be addition or deletion of tasks to or from a workflow, and changes in the sequence of tasks in a workflow.

In the workflow design of the previous section, routing information is stored in the `schema` object. Hence, a change in the workflow routing will lead to a new `schema` object. This new object might be generated in a number of ways, by transformation from an existing object or by design from scratch. The creation of this new `schema` object is not of interest here, we only consider different schema evolution policies given new or modified schema objects.

The subject of schema evolution is discussed extensively in [8]. The authors give a number of different policies to deal with activities in an evolving schema. The goal of these policies is to gracefully handle ongoing activities, that follow a schema that is modified. In brief, the following policies are identified:

Abort All activities following the old schema are aborted and restarted following the new schema.

Flush Ongoing activities are completed according to the old schema, while new activities are started following the new schema.

Progressive policies In these policies, ongoing activities are upgraded to a new schema without restarting.

To cater for the *Abort* policy, we must provide a number of facilities in the different objects. First, the `job` object must provide an action to abort its activities. This action must roll the object back to the state it was in, when it started. Since a DEGAS object contains its complete history, this is relatively easy to implement. A workflow, however, also has effects in the real world. Since agents are responsible for the interactions with the real world, they also provide the compensating actions. Roll back of actions is discussed in the next section.

Since routing information is communicated to the `job` object as late as possible, this rolling back can be completely transparent to the `job` object. After each task, the `job` object requests its next task. Instead of answering with the next task to complete the activity, the `schema` object replies with the next compensating task to roll the `job` object back to its initial state.

The *Flush* strategy is very easy to implement in any system. In our design, every `job` has a relation to a `schema` object. If a new `schema` project is created for an activity, the `job` object that are already being processed simply keep their relation with the old `schema` object. If a new `job` object is entered into the activity, it gets a relation with the new `schema` object.

The term *progressive policies* covers a number of different policies, which have in common that an activity is finished following a modified schema without a complete rollback of the old schema. The modified schema may or may not be the same as the new schema. A `job` can be switched over to the new schema, if the completed part of the old schema conforms to the new schema. In this case, we can simply change the `schema` object in the `jobflow` relation to the new schema.

Other progressive policies involve a special transition schema, that is only used to complete ongoing activities. A transition schema can implement a number of different methods. For example, it can contain a partial rollback, to get the job in a state, where its completed work conforms to the new schema. Another possibility is to append some special tasks at the end of the flow in order to get the same result as produced by the new schema. This approach is especially useful in manufacturing, where it can be used to retrofit the product with a modification.

All these approaches imply that a `schema` object, containing this transitional schema, is created. Since a `job` object does not contain any advance information about its routing, the change of schema can be enacted by a simple change of `JobFlow` relation. This is a distinct benefit of the DEGAS minimal information principle.

This approach also facilitates a final, truly ad-hoc, way of dealing with schema evolution. We can relate a `job` object to an interactive `schema` object, that prompts the workflow administrator for the next task on completion of each task in the activity. This might be useful for cases, where we only have a small number of `job` objects needing a transition schema.

6.2 Undoing tasks

In order to abort jobs or to apply progressive policies to jobs, we need the ability to roll back tasks. Aborting a task means that we have to reinstate the initial state of the `job` object. Some progressive policies may involve a partial rollback to a previous state. Here, we look at the problem of rolling back tasks in the workflow design discussed in this paper.

Previous states of a DEGAS object are stored as part of its history together with the actions that brought the object into that state. Hence, all information to bring back the `job` object to its original state is found in the object itself. Rolling back the actions with regard to the tasks that were executed, is basically a question of removing these actions from the history of the object. Since the current state is the latest state in th history, the current state of the object will then be automatically set to the state before

Removing actions from the history of an object only rolls back changes in the object itself, it does not undo the effects of interactions with the environment. The interactions with the physical environment of the workflow application are through the `agent` objects. In addition, the `job` object might have relations with other objects than the `agent` and `schema` objects. Hence, the rollback is a responsibility of both partners in the `TaskExecution`. As a consequence a workflow designer must provide *compensation* for each phase. The `job` objects cannot distinguish these compensating tasks from ordinary tasks. If a `job` follows a transitional schema, the schema object will simply give compensating tasks first as successor tasks.

Compensating tasks are analogous to the concept of compensating transactions used for sagas [9]. The difference is, that we do not need to roll back a transaction completely to its start, but that we can roll back only part of its actions. Hence, we relax the requirement on a saga with sub-transactions $T_1, \ldots, T_n$ with compensating transactions $C_1, \ldots, C_n$, that we execute either:

$$T_1; T_2; \ldots; T_n$$

or

$$T_1; \ldots; T_j; C_j; \ldots; C_1$$

for some $1 \leq j \leq n$. Instead we have a partial order between tasks, that may be extended. Suppose we have a partially ordered set of tasks T, where each $T_i \in T$ has an associated compensating action C_i. In addition, we have a function $Compensate(\tau)$, that yields the compensating task of task τ. Please note that a task undoes its compensating task, so the compensation of a task itself is its compensating task and $Compensate(Compensate(\tau)) = \tau$ Then, we have the following requirements on two subsequent tasks:

$$T_i; T_j \quad T_i < T_j \wedge \nexists t \in T : T_i < t < T_j$$

$$T_i; C_j \quad C_j = Compensate(T_i)$$

$$C_i; C_j \quad Compensate(C_i) > Compensate(C_j) \wedge \nexists t \in T : Compensate(C_i) > t > Compensate(C_j)$$

$$C_i; T_j \quad \exists t \in T : t < Compensate(C_i) \wedge t < T_j \wedge \nexists s \in T : s > t \wedge s < Compensate(C_i) \wedge s < T_j$$

6.3 Changing Task Execution

The other main source of evolution in a workflow lies in the way tasks are executed. In our design, the execution of tasks is a concern separated from routing, since tasks are executed by `agent` objects. The interaction between `agent` and `job` objects is specified in the `TaskExecution` relation. As a consequence, changes in task execution are easily separated.

A new way of executing a task might be completely transparent, in the sense that no additional information is needed from the `job` object. In this case, the only changes necessary are in the `agent` object. Hence, a `job` entering the execution of the changed tasks gets the same relation with the modified `agent` object. If the change in task execution requires additional interaction between `job` and `agent` objects, the `TaskExecution` relation and its associated addons will also need to be modified.

Whatever the type of change, the modularisation of workflow in this paper guarantees that a task is always executed according to the latest version. This is achieved by separating task execution from the jobs, so that a `job` object gets the necessary information as late as possible. Again this is an application of the DEGAS minimality design principle.

7 Conclusion

In this paper, we discussed an approach to designing an active database in DEGAS. We described how database design in DEGAS is guided by two principles: Minimality of information and maximality of encapsulation. The minimal information principle is a guideline for the designer, which is facilitated by the relation and addon mechanisms. The maximal encapsulation principle is part of the DEGAS model. An advantage relative to other active databases is the use of the ordinary object-oriented notions for modularisation of the rulebase. Hence, we do not need additional concepts, such as a rulebase.

The DEGAS design guidelines were applied to the example of workflow management. Although active databases are in general well suited for the implementation of workflow management, there is a need for clearly modularised active database designs for this application. We have shown that the DEGAS design guidelines lead to a design with clearly separated responsibilities of the different objects. Furthermore, we have shown that it facilitates a straightforward implementation of workflow evolution strategies.

References

[1] W.M.P. van der Aalst, K.M. van Hee, and G.J. Houben. Modelling workflow management systems with high-level Petri nets. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Proceedings of the Second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31-50, 1994.

[2] J.F.P. van den Akker and A.P.J.M. Siebes. DEGAS: A temporal active data model based on object autonomy. Technical Report CS-R9608, CWI, Amsterdam, The Netherlands, 1996. Available through WWW (`http://www.cwi.nl/~vdakker/`).

[3] Johan van den Akker and Arno Siebes. DEGAS: Capturing dynamics in objects. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Advanced Informations Systems Engineering - Proc. of CAiSE'96*, pages 82-98, Heraklion, Crete, Greece, May 1996. Springer. LNCS 1080.

[4] Johan van den Akker and Arno Siebes. Object histories as a foundation for an active OODB. In R. Wagner and H. Thoma, editors, *Proceedings of the 7th International Workshop on Database and Expert Systems Applications (DEXA'96)*, pages 2-8, Zürich, Switzerland, 1996. IEEE Computer Society.

[5] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1990.

[6] Elena Baralis, Stefano Ceri, and Stefano Paraboschi. Modularization techniques for active rules design. *ACM Transactions on Database Systems*, 21(1):1-29, 1996.

[7] F. Casati, C. Ceri, B. Pernici, and G. Pozzi. Deriving active rules for workflow enactment. In R. Wagner and H. Thoma, editors, *Proceedings of the 7th International Conference on Database and Expert Systems Applications (DEXA'96)*, pages 94-115, Zürich, Switzerland, 1996. Springer. LNCS 1134.

[8] F. Casati, C. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. In Bernhard Thalheim, editor, *Conceptual Modelling - Proceedings of the 15th ER Conference (ER'96)*, pages 438-455, Cottbus, Brandenburg, Germany, 1996. Springer.

[9] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proc. of the 1987 SIGMOD Intl. Conf. on the Management of Data*, pages 249-259, 1987.

[10] Stella Gatziu, Andreas Geppert, and Klaus R. Dittrich. Integrating active concepts into an object-oriented database system. In Paris Kanellakis and Joachim W. Schmidt, editors, *The Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data*, pages 399-415. Morgan Kaufmann, 1991.

[11] H. Herbst, G. Knolmayer, T. Myrach, and M. Schlesinger. The specification of business rules: A comparison of selected methodologies. In A.A. Verrijn-Stuart and T.-W. Olle, editors, *Methods and Associated Tools for the Information System Life Cycle*, pages 29-46, Amsterdam, the Netherlands, 1994. Elsevier.

[12] Heinrich Jasper, Olaf Zukunft, and Helge Behrends. Time issues in advanced workflow management applications of active databases. In *Active and Real-Time Database Systems (ARTDB-95)*, Workshops in Computing, pages 65-81. Springer, 1995.

[13] P. Lang, W. Obermair, and M. Schrefl. Situation diagrams. In R. Wagner and H. Thoma, editors, *Proceedings of the 7th International Conference on Database and Expert Systems Applications (DEXA'96)*, pages 400-421. Springer, 1996. LNCS 1134.

[14] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall International, 1988.

[15] G.M. Nijssen and T.A. Halpin. *Conceptual schema and relational database design : a fact oriented approach*. Prentice-Hall, New York, USA, third edition, 1990.

[16] James Rumbaugh et al. *Object-oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, USA, 1991.

[17] Marek Rusinkiewicz and Amit Sheth. Specification and execution of transactional workflows. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 29, pages 592-620. Addison-Wesley, 1995.

[18] Jennifer Widom. *The Starburst Rule System*, chapter 4 in [19].

[19] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, CA, USA, 1995.

[20] Roel Wieringa, Wiebren de Jonge, and Paul Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61-83, 1995.

[21] Workflow Management Coalition. Terminology and Glossary WFMC-TC-1011 2.0. Available through WWW at `http://www.aiai.ed.ac.uk/WfMC/`, June 1996.

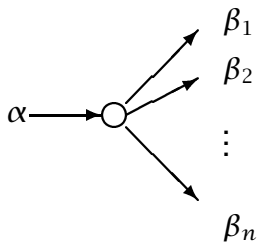
Appendix:Workflow routing elements

The Workflow Management Coalition identifies five different routing elements:

1. AND split
2. AND join
3. OR split
4. OR join
5. Iteration

A split means that a task has multiple successors, while a join means multiple predecessors. AND means that all successors or predecessors are involved. Likewise OR means that only one of the successors or predecessors is involved.

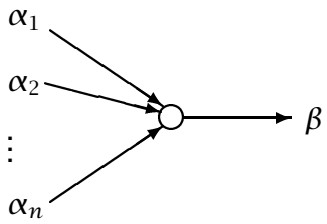
An AND split means that a task has a number of successors, which are all started up simultaneously. In the following picture, this means that the tasks $\beta_1, \beta_2, \ldots, \beta_n$ are started simultaneously after α has finished.



The associated conditions for the execution of the tasks β_1 to β_n are:

$\alpha \in Completed(A)$	**On** End(α)
$\wedge$	**do** Start(β_1)
$Inhib(\beta) \cap Completed(A) = \emptyset$	$\vdots$ $\quad$ $\vdots$
$\wedge$	**On** End(α)
$Precond(\beta_i)$	**do** Start(β_n)

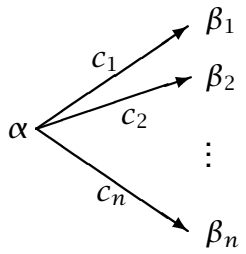
An AND-join specifies that a task may start only if a number of preceding tasks have all been completed. Here, β is started after $\alpha_1, \alpha_2, \ldots, \alpha_n$ have all been completed.



The condition for the start of task β is:

$Pred(\tau) \subseteq Completed(A)$	**On** $\|_{i=1...n}$ End(α_i)
$\wedge$	**do** Start(β)
$Inhib(\tau) \cap Completed(A) = \emptyset$	
$\wedge$	
$Precond(\tau)$	

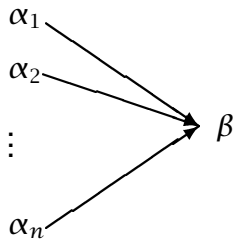
The previous two routing elements specified tasks that executed in parallel. We can also specify the selection of a subset of successor tasks, so that not all successor tasks need to be executed.



If we formalise the workflow in terms of preconditions, predecessors, successors, and inhibitors, this case is not different from the AND-split. The condition for the start of each β_i again is:

$\alpha \in Completed(A)$	**On** End(α)
$\wedge$	**if** C_1
$Inhib(\beta) \cap Completed(A) = \emptyset$	**do** Start(β_1)
$\wedge$	$\vdots$ $\vdots$
$Precond(\beta_i)$	**On** End(α)
	if C_n
	do Start(β_n)

An OR-join is different from an AND-join in that only one of the predecessors needs to be completed to start the task. In the following picture, β can be started on completion of either α_1, α_2, or α_n.

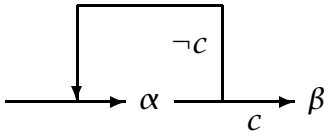


Hence, the set of predecessors of β needs not be a subset of the set of completed tasks:

$Pred(\beta) \cap Completed(A) \neq \emptyset$
$\wedge$
$Inhib(\beta) \cap Completed(A) = \emptyset$
$\wedge$
$Precond(\beta)$

On End(α_1)
do Start(β)
⋮ ⋮
On End(α_n)
do Start(β)

The final routing element defined by the WfMC is iteration. Iteration means that a task α is repeated until a condition c is satisfied. If c is satisfied, the activity proceeds with the next task β.



Iteration can be formalised as an OR-split, with α as a successor to itself with $\neg c$ as a precondition and with c as a precondition for β.

To start α:

$\alpha \in Completed(A)$
$\wedge$
$\neg c$
$\wedge$
$Inhib(\alpha) \cap Completed(A) = \emptyset$

On End(α)
if C
do Start(α)

To start β:

$\alpha \in Completed(A)$
$\wedge$
c
$\wedge$
$Inhib(\beta) \cap Completed(A) = \emptyset$

On End(α)
if $\neg C$
do Start(β)

Appendix: DEGAS source of the workflow example

```
Object Order
Attributes
    number : integer
    price : real
    currentTask : number
Methods
Lifecycle
Rules
EndObject
```

```
Addon InventoryBeingChecked
extends RoutedBy
Attributes
Methods
    EndInventoryCheck = {
        nextPhase
    }
Lifecycle
    Extend(InventoryBeingChecked);
        EndInventoryCheck;Delete(InventoryBeingChecked)
Rules
    On Extend(InventoryBeingChecked)
        do InventoryCheck.Start(number)
    On EndInventoryCheck
        do InventoryCheckClass.terminateRelation
EndObject
```

```
Object InventoryCheck
Relation InventoryControl, Order
Attributes
Methods
    Start(number : integer) = {
        InventoryControl.request(number)
    }
    Finish() = {
        Order.EndInventoryCheck
    }
Lifecycle
    Start;Finish
Rules
EndObject
```

Object InventoryControl
Attributes
Inventory : integer
Reserved : integer
Methods
reserve(number:integer) = {
Reserved = Reserved + number
}
newstuff(number:integer) = {
Inventory = Inventory + number
}
Lifecycle
reserve*
newstuff*
Rules
EndObject

Addon CheckingOrder
extends InventoryControl
Attributes
InventoryCheck : Oid
noOfItems : integer
Methods
request(number : integer, dest : oid) = {
noOfItems = number
}
getit(number : integer, dest : oid) = {
reserve(number)
}
Lifecycle
request*;getit
([sender==InventoryCheck]request)*
([sender==self]request)*
Rules
On request(number,dest)
if number ≤ Inventory - Reserved
do getit(number,dest)
On request(_,dest);newstuff;¬getit(_,dest)
do request(noOfItems,dest)
On getit(number,dest)
do dest.Finish
EndObject

Addon RoutedBy
extends Order

```
Attributes
    jobflow : Oid
    CurrentTask : string
Methods
    nextPhase(current:string) = {
        jobflow.whatNext(current)

Lifecycle
    Extend(RoutedBy);nextPhase*;Delete(RoutedBy)
Rules
EndObject
```

```
Addon Router
extends Schema
Attributes
    jobflow : Oid
Methods
    successor(task:string) = {
        return succ(task)
    }
Lifecycle
    successor*
Rules
EndObject
```

```
Object JobFlow
Relation Schema, Job
Attributes
Methods
    whatNext(task : string) = {
        succ = Schema.successor(task)
        Job.Extend(succ)
    }
    replaceSchema(newSchema:oid) = {
        Schema.delete(Router)
        Schema = newSchema
        newSchema.extend(Router)
    }
Lifecycle
    (sender==Job)whatNext*
    replaceSchema*
Rules
EndObject
```