DEGAS: A DATABASE OF AUTONOMOUS OBJECTS

JOHAN VAN DEN AKKER AND ARNO SIEBES

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

October 15, 1996

Abstract — In this paper we introduce DEGAS (Dynamic Entities Get Autonomous Status), an active temporal data model based on autonomous objects. The natural combination of active and temporal databases is discussed. The active dimension of DEGAS means that we define the behaviour of objects in terms of production rules. The temporal dimension means that the history of an object is included in the DEGAS data model. Further novel features of DEGAS are the encapsulation of the complete behaviour of an object, both potential and actual. Thus, DEGAS combines dynamic and structural specifications in one model. In addition, DEGAS allows easy evolution of object capabilities through a clear distinction between inherent types and capabilities that can be acquired and lost. This addon mechanism makes DEGAS very suitable as a formalism for role modelling. Finally, the rule model in DEGAS is both simple, through the use of finite automata, and general, because it allows different strategies for dealing with constraints and reacting to events in other objects.

Key words: active databases, objects, roles

1. INTRODUCTION

It is widely recognised that information systems (IS) modelling should include both static and dynamic aspects of their universe of discourse. To facilitate effective IS design, integration of these aspects must be supported in all phases of the IS development process. This includes implementation platforms for informations systems, database management systems, that have traditionally focussed on the static side of information systems.

Encapsulation of methods and data in object-oriented databases is a step forward in the integration of the dynamic and static parts of an application. Active databases [26, 11] integrate another dynamic element into databases, viz., production rules. Originally rules were introduced to deal flexibly with constraints in a database. Much wider use, however, has been found for them. In fact, it is possible to encode the entire dynamics of an information system as rules in an active database system. In DEGAS we unify both approaches in one active, object-oriented data model. Since both active and temporal databases are concerned with the evolution of an object over time, we also include a temporal dimension in DEGAS for an effective unification.

The specification of a DEGAS object has a static part, attributes, and a dynamic part, described by methods, production rules, and lifecycles. Thus, DEGAS achieves the encapsulation of the complete dynamic aspect of an information system. This total encapsulation is a means to achieve object autonomy. Systems built of autonomous components are necessary for the development of networked information systems across multiple organisations.

Roadmap

In this paper, we first introduce the key concepts of DEGAS. Then, we give a further feel of DEGAS by showing an example of its use. After that, we define the state of an object in DEGAS, which means a discussion of the history of an active database. Next, we formalise lifecycles and rules in DEGAS objects. After a discussion of the DEGAS relation and addon mechanism, we conclude with a broader perspective on a database of autonomous objects.

2. MAIN CONCEPTS

The fundamental notion of the DEGAS model is the object. It has structure and behaviour. The structure of an object is determined by the attributes. The behaviour of an object has three components: methods, lifecycles, and rules. Methods specify *what* an object can do. The lifecycles specify what methods the object is willing to execute in a certain context, by specifying sequencing and preconditions of method execution. A rule states *when* an object will execute a given action as far as can be modelled within the system. In other words, rules specify actual actions to be executed in certain situations, described in terms of events and object states.

Thus, methods and lifecycles specify *potential* behaviour of an object, whereas rules describe *actual* behaviour. Traditionally only potential behaviour is specified, whereas DEGAS objects also contain their actual behaviour as far as that can be pre-determined.

In DEGAS relations are modelled as objects. Thus, we have a place for data and behaviour of a relation. A more abstract motivation of this objectification is that a relation is a kind of contract, a view also found in, for example, NIAM [19].

The class of an object defines the intrinsic capabilities, i.e., attributes, methods, rules and lifecycles. Addons are used to model transient capabilities. That is, addons are used to define capabilities that can be added and deleted from an object dynamically. Addons can be likened to roles and are DEGAS' only mechanism for object specialisation.

3. AN EXAMPLE

Challenging applications to model are those with high dynamics. An application with fast changing data and rapidly evolving relations is the stock market. New data emerges constantly in the form of buying and selling orders, economic news items through newsreels etc. Both new and historical data influence the behaviour of the parties in the market. In order to introduce the concepts of DEGAS, we model this example. Our example is a simplification of the system used in the Netherlands.

Let us briefly describe the example in more detail. Companies are owned by persons through shares. A person can buy shares and sell them again. He can subscribe to a newspaper to get news about the companies he is interested in. The buying and selling of shares goes through a marketmaker. If a person wants to buy or sell, he informs the marketmaker. Periodically, the marketmaker determines the price that balances supply and demand. Buying and selling orders that agree with this price are fulfilled.

We start to model this example with the marketmaker. The marketmaker matches supply and demand for his market. This means that the actions he can execute are to accept buying and selling orders and to try to match these. This is specified by the following DEGAS definition of the marketmaker object's attributes and methods. The methods in this object only contain actions to engage in a relation or to extend the object with an addon.

```
Object Marketmaker
Attributes
    currentPrice : real
Methods
    takeSellOrder = {
        SupplyClass.initiate
    }
    takeBuyOrder = {
        DemandClass.initiate
    }
    makeMarket = {
        SupplyDemandAddon.extend
    }
```

This defines possible actions the object may execute, but we know more about the actions of a marketmaker. Therefore, an object includes a lifecycle description. Lifecycles are specified by guarded basic process algebraic expressions [5] with method names as basic actions. The following operators are used in lifecycle specification:

Sequence	$A; B$	A followed by B
Choice	$A + B$	A or B
Repetition	A^*	One or more times A
Merge	$A \| B = A; B + B; A$	A and B in parallel

Each basic action can be prefixed by a condition as a guard as follows.

$$[\langle Condition \rangle]\langle Methodname \rangle$$

It can be used to express a precondition of a method, or to restrict access to the method by other objects. To this purpose, we can refer to a variable *sender*, which contains the object sending the method call.

The lifecycle of a marketmaker consists of taking buying and selling orders. If these are both present, he is allowed to match supply and demand.

Lifecycles
((takeSellOrder* || takeBuyOrder*);makeMarket)*

The specification of the actual execution of actions by a DEGAS object is given by rules. The behaviour of a marketmaker is to extend himself, if he has both supply and demand relations. This is specified by the following rule, that completes the definition of the marketmaker object.

Rules
On (takeSellOrder||takeBuyOrder) **do** makeMarket
EndObject

This rule extends the Marketmaker with the SupplyDemandAddon. This addon contains a rule that periodically triggers the necessary actions to clear the market.

In our example a person can buy shares. To do this he places a buying order. If this order can be met by supply in the market, he will actually buy the shares. If it is unsuccessful, a cancellation will be the result. In addition to buying shares, a person can take a subscription to a newspaper in order to obtain information. If he owns shares and also reads a newspaper, he uses the information from the newspaper to influence his decisions about his shares. This is specified in the person object given in Figure 1

The person and marketmaker objects have relations specifying that the object engages in a relation. Relations in DEGAS are objects themselves. A relation object can have the same capabilities as an ordinary object. For example, a share is modelled as an ownership relation between a person and a company. In the relation object, the partners in the relation are present as implicit attributes by the *Relation* line. Other information it contains, is the price of the share when it was bought. The definition of the share relation object shows the use of guard conditions in the lifecycle. The action after a condition can only be executed, if the condition is satisfied. In the Share relation object given in Figure 2, guards are used to restrict access to its methods.

A person object does not have the capability to deal with the share relation built-in. Instead it acquires these when it engages in this relation. This is represented by the shareholder addon in Figure 3. An addon defines a temporary specialisation of an object, which is lost when the relation is terminated. In this example, a person who is also a shareholder gains capabilities to sell the shares again.

The *SupplyClass.initiate* action in this addon specification also occurred in the specification of the Marketmaker object. A call to an *initiate* method is done by an object to express its wish to engage in a relation. Since the relation object does not exist at this time, *initiate* is a method of the relation class object. In this case a Shareholder object will send an *initiate* call to the Supply

```
Object Person
Attributes
    name : string
    birthday : time
    birthplace : string
Methods
    tryToBuy(company:string, number:integer, maxPrice:real) = {
        DemandClass.initiate(company,number,maxPrice)
    }
    readPaper(paper:string) = {
        SubscriptionClass.initiate(paper)
    }
    useNews = {
        InformedOwnerAddon.extend()
    }
Lifecycles
    (tryToBuy)*
    ((extend-Shareholder||extend-InformedPerson);useNews)*
Rules
    On (extend-Shareholder||extend-InformedPerson) do useNews
EndObject
```

Fig. 1: The Person object

```
Object Share
Relation Person, Company
Attributes
    buyPrice : real
    currentPrice : real
    value : real
Methods
    transferOwnership(newOwner:oid,price:real) = {
        Person = newOwner
        buyPrice = price
    }
    payDividend(div:real) = {
        value = value + div
    }
Lifecycles
    ([sender==Person]transferOwnership)*
    ([sender==Company]payDividend)*
EndObject
```

Fig. 2: The Share relation object

```
Addon Shareholder
Extends Person
Attributes
    share : oid
Methods
    tryToSell(company:string, number:integer, minPrice:real) = {
        SupplyClass.initiate(company,number,minPrice)
    }
    Sell(buyer,price) = {
        share.transferOwnership(buyer,price)
        Supply.drop
    }
    cancelSupply = {
        Supply.drop
    }
Lifecycles
    (tryToSell;(Sell+cancelSupply))*
EndAddon
```

Fig. 3: The Shareholder addon

class object. In response it will send a *takeBuyOrder* message to the marketmaker to ask, if it is willing to accept the relation. As we can see in the specification of the Marketmaker object, it will respond with an *initiate* call to express its agreement. The Supply class object will then proceed to instantiate the relation.

As we can see above in the specification of the Person object, an addon can also be used to link two relations. In our example, the information a person reads in the paper will influence his decisions as a shareholder. This is achieved by extending the person with a further addon, if he owns shares and reads a newspaper. In Figure 4, we give the specification of the InformedPerson addon, that extends a person who has a subscription to a newspaper.

The rule specification in these addon definitions shows the use of time in DEGAS. Historical values of attributes can be referenced by a time parameter. Likewise, we can refer to the timestamp of an event. The addon specification in Figure 5 gives an example of how the informed owner of shares would deal with bad news. This addon extends a person, if it has both the Shareholder and the InformedPerson addons. Therefore, the *extends* specification gives two original object names. Please note, that this does not introduce a form of multiple inheritance. It simply specifies, what the addon may assume to be present in the object it extends.
The diagram in Figure 6 shows the complete model of the stock market example. In this picture, large boxes represent objects and small boxes represent addons. The dashed boxes are relation objects. The outgoing arrows from relation objects indicates the partners in the relation, they do not imply any arity constraint on the relation.

4. INTEGRATION OF TEMPORAL FUNCTIONALITY

Through the inclusion of Event-Condition-Action rules in objects, DEGAS is an active database system. Although the natural combination of temporal and active databases has been suggested by different authors [10, 26], there are very few active data models that incorporate the history of a database. Work on the temporal specification of rules has been done by Sistla and Wolfson [24]. This approach is based on temporal logic. It focusses only on the condition of the rules, leaving out the event specification of a rule. Another approach is that in [12], which gives a model for derived data in a temporal database.

This section describes the temporal element of DEGAS, which is formed by the history of an

```
Addon InformedPerson
Extends Person
Attributes
    subscription : Oid
    transactionPrice : real
Methods
    goodNews(company : string) = {
        transactionPrice = subscription.priceAdvice(company)
    }
    badNews(company : string) = {
        transactionPrice = subscription.priceAdvice(company)
    }
Lifecycles
    ([sender==subscription]goodNews*)
    ([sender==subscription]badNews*)
    (ExtendInformedPerson;DropInformedPerson)*
Rules
    On goodNews(company)(t1);goodNews(company)(t2)
    if t2 - t1 ≤ 7 days
    do tryToBuy(company,transactionPrice)
EndAddon
```

Fig. 4: The InformedPerson addon

```
Addon InformedOwner
Extends InformedPerson,Shareholder
Attributes
    Key : P⟨ Subscription : Oid, Share : Oid ⟩
Lifecycles
    ExtendInformedOwner*
    DropInformedOwner*
Rules
    On badNews(company)(t1);badNews(company)(t2)
    if (t2 - t1) ≤ 7 days && transactionPrice(t2) ≤ transactionPrice(t1)
    do tryToSell(transactionPrice)
    On goodNews(t1);badNews(t2)
    if t2 - t1 ≤ 7 days && transactionPrice(t1) == max(transactionPrice, t1, t2)
    do tryToSell(transactionPrice)
    On DropShareHolder do DropInformedOwner
    On DropSubscription do DropInformedOwner
EndAddon
```

Fig. 5: The InformedOwner addon

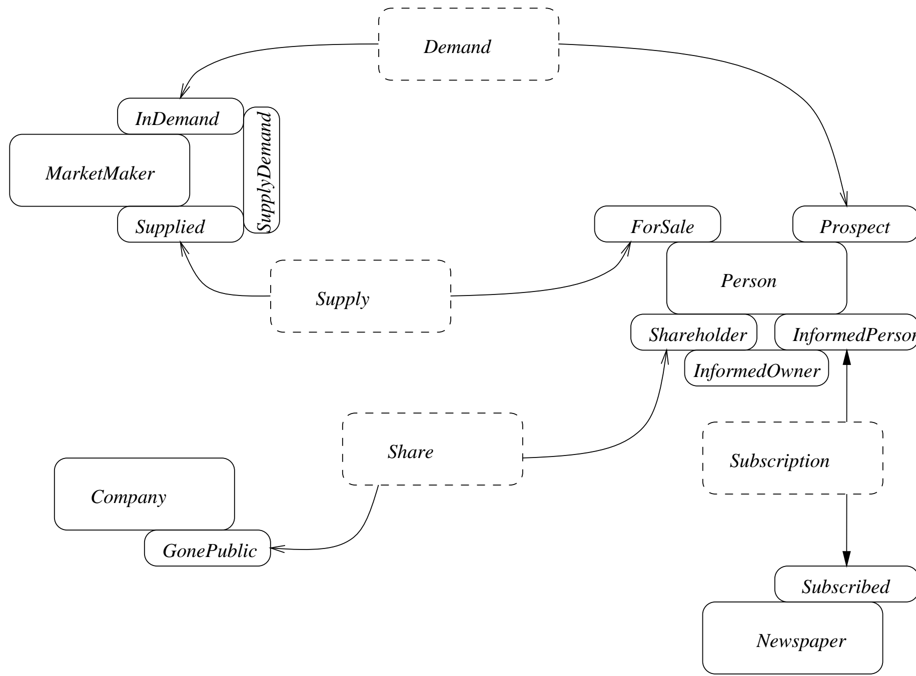

Fig. 6: The DEGAS model for a financial market

object. First, we will investigate the link between active and temporal databases. Then, we will look at the way temporal functionality is integrated in DEGAS

4.1. Time in Active Databases

The temporal element in an active database is mainly introduced through the event specification in rules. The event specification may be a complex event expression composed of multiple basic events, such as method calls [9, 13] and time events [9, 16, 13]. Since rule definitions specify sequences of events over time, an active database has an inherent temporal element, as observed by Dittrich [10] and Widom and Ceri [26].

We can also see this by looking into rule triggering in more detail. In order to detect complex events, we need to store the basic events occurring in the database. These make up an event queue or event pool. Since a complex event expression usually specifies a sequence of events, the record of basic events must store information about the order in which events occurred.

This inherent temporal element in active databases gives rise to the question of the relation to databases that keep historical data. To that end, we examine what temporal data needs to be stored in an active databases. Not surprisingly, this depends on the rule language.

Many active databases include time in an event expression. This can be in relative form, such as "5 days after event A" or absolute such as "every day at midnight". Orthogonally, we can put time in event specifications in different ways. We can add a time parameter to all events or we can have explicit time events in the event specifications. The latter choice will make a difference in the way we check the temporal part of the rule specification. In the former case, we can check temporal conditions in the condition of the rule. In the latter, the time events are included in the event detection mechanism.

Since most active database management systems offer the possibility to specify parameters of events, we also need to store the parameters of a method call in addition to the time it occurred. In this way a rule can be triggered on method calls only for certain parameter values. For example, we may have a rule on a bank account that is only invoked if a debit action of more than 1000 guilders is executed.

Every extension of event specification in the definition of rules beyond single basic events

necessitates a record of the recent history of the database. If we wish to offer all facilities described above in an active DBMS, i.e., time in event specifications and parameters to events, we have to store all method calls with their parameters and timestamps. It is obvious that we can reconstruct all historical states of the database, if we have all state transitions in the form of method calls. Hence, it is a small step from an active database to a historical database.

A historical database is a restricted form of temporal database. Temporal databases [25] record data relative to time. A full temporal database has two temporal dimensions. Valid time denotes the time a value held in the real world. Transaction time denotes the time a value was entered into the database. The combination of these two dimensions allows us to alter data retrospectively, for example, to correct errors. Historical databases are temporal databases with only one temporal dimension. In other words, a historical database only records the states a database went through over time.

4.2. The History of a DEGAS Object

DEGAS is a historical database by recording the complete history of an object, represented as a sequence of snapshot states. In temporal database research [25], the term snapshot state denotes the state of an object at a point in time when we abstract from the temporal dimension [18].

For the definition of a snapshot state, we first need to define the type of an object at a given time. The type of an object is defined by its attributes. Attributes in a DEGAS object are typed. From a number of simple types, like integers, reals, strings, and object identifiers, additional types are formed using set and tuple constructors. The type system underlying DEGAS follows Cardelli [7] and Balsters [6].

The underlying type of an object is a tuple type containing the attributes. The underlying type of an object definition contains at least its own identifier *this*. An operator *Type(D)* can be applied to an object definition to obtain the underlying type of the object. For example, the underlying type of the share object from Section 3 is

$$Type(\text{share}) = \langle this : Oid, currentPrice : real, person : oid, company : oid \rangle$$

A DEGAS object is brought from one state into another by the execution of a method. The actions that can be included in a DEGAS method specification are assignment to attributes and calls to other methods. In addition, we can map a method over a set, i.e., a method can be applied to all elements of a set.

A snapshot state records the time the object came in this state, the type of the object, a valuation for the type and the method call that brought the object into this snapshot state. More formally, a snapshot state of an object O is a quadruple $\langle t, \tau, I(\tau), MC \rangle$, where t is a timestamp giving the start time of the validity of this state, τ the type of O at time t, $I(\tau)$ is the valuation of τ and MC is a method call, which consists of a method name and a parameter list. The inclusion of the type of an object in the snapshot state is motivated by the evolution over time of an object's type through the addon mechanism. For example, we might have a Person object o , that is extended by a Shareholder addon at time t. Then, before t the type of o is:

$$\langle this : oid, name : string, birthday : time, birthplace : string \rangle$$

After extensions by a Shareholder addon, the type of o is:

$$\langle this : oid, name : string, birthday : time, birthplace : string, share : oid \rangle$$

State History

The state history of a DEGAS object records the snapshot states the object went through during its existence. This means that a state history SH is a sequence of snapshot states:

$$SH = SH(0); SH(1); \ldots; SH(n)$$

where $\forall i, 0 \leq i \leq n-1 : t_i < t_{i+1}$. This definition of an object history is largely similar to that found in [14]. The main difference is that a DEGAS object history deals directly with DEGAS methods calls

and attributes, instead of the more abstract notions of actions and input and evaluation attributes in [14]. As an example, the following is a piece of the history of a share object:

$$\begin{aligned}
&\vdots \\
(13:00:00, &\langle currentPrice : real, person : oid, company : oid, \rangle, \\
&\langle currentPrice = 54.25, person = Johan, company = Philips\rangle, \\
&transferOwnership(Johan, 54.25)) \\
(13:02:00, &\langle currentPrice : real, person : oid, company : oid\rangle, \\
&\langle currentPrice = 55.25, person = Arno, company = Philips\rangle, \\
&transferOwnership(Arno, 55.25)) \\
&\vdots
\end{aligned}$$

The information in the snapshot state is valid from the given time until the time given in the next snapshot state. The last snapshot state gives the information valid at the current time.

Historical values of attributes are accessed in DEGAS through the addition of a time parameter. This can be used, for example, in the condition of a rule:

Rules
On share.newPrice
if price($T_{now} - 15$ min) - price(T_{now}) > 10
do tryToSell

The meaning of this rule is that we try to sell shares after a new quotation, if the current price is more than 10 guilders less than the price 15 minutes ago.

Event History

Lifecycles and the event expressions of rules are checked using a projection of the state history, the event history. It only contains timestamp-method call pairs. If we have a state history $SH = SH(0); SH(1); \ldots; SH(n)$, then the event history EH is the sequence $EH(0); EH(1); \ldots; E(n)$ of time-event pairs, where:

$$\begin{aligned}
\forall i, SH(i) = \langle t_i, \tau_i, I(\tau_i), MC_i\rangle : \\
EH(i) \stackrel{def}{=} \langle s_i, e_i(p_1, \ldots, p_m)\rangle, \\
s_i = t_i \wedge e_i(p_1, \ldots, p_m) = MC_i
\end{aligned}$$

The example state history above gives us this event history:

$$\begin{aligned}
&\ldots \\
&\langle 13:00:00, transferOwnership(Johan, 54.25))\rangle; \\
&\langle 13:02:00, transferOwnership(Arno, 55.25))\rangle \\
&\ldots
\end{aligned}$$

5. THE BEHAVIOUR OF A DEGAS OBJECT

The history of a DEGAS, defined in the previous section, is the central element in the formalisation of its behaviour. In this section, we see how sequencing of method calls is restricted by the lifecycles of an object. In addition, rule triggering is defined in terms of the history of an object.

Lifecycles

A method call can be executed on an object, if it satisfies the object's lifecycle. The semantics of lifecycles is defined using process algebra [5]. Suppose we have defined the following set of lifecycles on an object O.

Lifecycles
C_1
C_2
$\vdots$
C_n

This means that the execution of methods on O must follow the process

$$C = C_1|C_2|\dots|C_n$$

with communication function γ defined by: $\forall\alpha \in \mathcal{M} : \gamma(\alpha, \alpha) = \alpha$, where $\mathcal{M}$ is the set of methods of O. In process algebra a communication function γ specifies synchronisation between two processes. $\gamma(A, B) = C$ means that the actions A and B have to take place simultaneously and a replaced in the trace of the process by the single action C. For example, if we have the process $(A; B)|(C; D)$ and $\gamma(B, D) = E$, then a resulting trace might be: $A; C; E$. In practical terms the communication function defined for a DEGAS object means, that if an action occurs in more than one lifecycle, the execution of that action is a step forward in all lifecycles.

Lifecycles can be checked using finite automata. This follows from the fact that lifecycles are regular expressions. The transitions in this automaton are labelled with method names and conditions. If there is an appropriate transition available, a method call can be executed.

The formalisation is, that a method call $MC = m(q_1, \dots, q_k)$ is executed on an object O with state history $EH = EH(0); \dots; EH(n)$ at time t, iff the event history that would result from the execution of MC satisfies the lifecycle of the object. In process algebraic terms, we say that:

$$EH = EH(0); \dots; EH(n); MC$$

must be a trace of the process specified by the lifecycle. The resulting new state of the object is

$$SH' = SH; \langle t, M(m(q_1, \dots, q_k), I(\tau)), MC\rangle$$

Here $M(m(q_1, \dots, q_k), I(\tau))$ denotes the result of the execution of $m(q_1, \dots, q_k)$ on attribute valuation $I(\tau)$.

Rules

In the specification of rules, DEGAS follows the ECA format, originally introduced by Dayal [9] and now commonly accepted in the active database community. Rules are specified as an Event-Condition-Action (ECA) triple. If the event occurs and the database state satisfies the condition, the action is executed. This is specified in a DEGAS object as:

On ⟨*Event*⟩ **if** ⟨*Condition*⟩ **do** ⟨*Action*⟩

The event specification is a basic process algebraic [5] expression constructed from a set of method calls. The event expression of a rule differs from a lifecycle by the absence of guard conditions. In addition a negation operator can be used in the event specification of a rule.

Negation	$A; \neg B$	A not followed by B

A rule condition is a condition on the state of the object. The action is a method call, either local or to a method in another object.

A rule is triggered by an event occurring in the event history as a result of a method call. If we have an event history $EH = EH(0); \dots; EH(m)$ and a method call $MC = \mu(p_1, \dots, p_k)$ at a time t. Rule $R = \langle E, C, A\rangle$ is triggered at time t, if E parses the new event history $EH; \langle t, \mu(p_1, \dots, p_k), t\rangle$ correctly.

The presence of rules means that there are two possible sources of actions in an object. The first consists of method calls from other objects. The second is the execution of actions from triggered

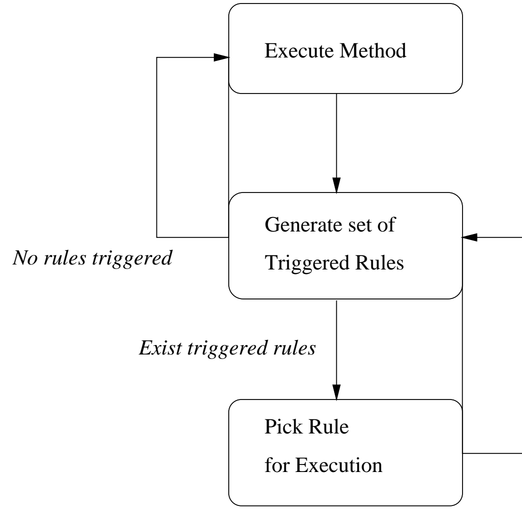

Fig. 7: The execution of an autonomous object

rules. Both are subject to the lifecycles specified on the object. This is reflected in the execution model of an object. Basically, an object first executes a method and then executes triggered rules until no more rules are triggered. This loop is depicted in Figure 7.

The execution of rules is a two phase process. During a method call, a set of rules that are triggered by that method call is built up. After a method has terminated, one rule is picked at random for execution from the set of triggered rules at random. If the condition of the rule is satisfied at the time it is picked, its action is executed. The rules in the triggered set that were not picked are dropped. During execution of a rule's action, a new set of triggered rules is constructed.

In more formal terms, we can define rule execution as follows. Given a triggered rule set $\mathcal{R}_\mu$ after the execution of method μ on an object O. The rule execution phase follows the algorithm:

1. $\mathcal{R} = \mathcal{R}_\mu$
2. A rule $R = \langle E, C, A \rangle$ is picked at random from $\mathcal{R}$.
3. If C is true and A satisfies the lifecycle of the object, A is executed. Otherwise, discard R and goto step 2.
4. Generate a new set $\mathcal{R}$ of rules triggered by the action of R.
5. If $\mathcal{R} \neq \emptyset$, then goto step 2.

The rule model ensures both simplicity and generality. The former originates from the fact that all event specifications can be checked using finite automata. The latter can be found in the different strategies that can be applied to the interaction of lifecycles and rules. If the action of a rule is not allowed by any lifecycle the moment it is triggered, we have two options. Either the action is tried again later, or the action is simply dropped. DEGAS offers both strategies through the negation operator $\neg$ in event expressions. If we want the action of a rule to be retried, the event includes the negation of the rule's action. For example, the marketmaker might have the rule that he *must* clear the market after he has determined a price:

Rules
  On determinePrice;$\neg$ clearMarket **do** clearMarket

Rules where only an immediate reaction is of interest, are, for example, those rules defining the reactions to news events in the InformedOwner addon in the previous section. In this case, the reaction is only useful if it is executed immediately.

A number of design issues are simplified by the DEGAS rule model. An example is the risk of non-termination, which is already undecidable for very simple rule languages [23]. In DEGAS this risk is taken explicitly by the designer by using the negation operator. Thus, he knows that a certain rule will be triggered again and again until it is executed. This assists in the identification of possibly problematic rule sets.

Other examples of object-based active database systems are HiPAC [9], SAMOS [13] and Chimera [8]. SAMOS and Chimera offer encapsulation of rules, but the object is not the exclusive location of rule definition. HiPAC treats rules as separate objects, thus separating part of the behaviour from the objects. The motivation given for this objectification of rules is that it allows easy run-time manipulation of rules. In other active databases, this is facilitated by the introduction of rule sets as manipulable units. In DEGAS this kind of manipulation is offered through the addon mechanism, which allows rules to be added to objects, i.e. activated, when necessary. Hence, DEGAS offers a modularisation of rules, while retaining the encapsulation of object behaviour.

6. RELATIONS AND ADDONS

Relations in DEGAS are objects themselves. Hence, the discussion of the elements of DEGAS objects above also applies to relation objects. The initiative for a relation comes from one of the partners. To this end it sends a message to the class object of that relation. Before the relation is established, there might be a number of conditions that need to be satisfied. These are checked by the relation class object. If the relation class object approves, it instantiates the relation object and instructs the partners to extend themselves with the appropriate addon.

An addon specifies an extension to an object's capabilities. It is a general purpose object specialisation mechanism. Hence, it is not tied to one particular form of specialisation, such as e.g. roles. An addon can be added to an object, if the object has a method to do this. This means that the object knows the name of the addon, but does not know anything about the contents of an addon. Thus, changes in an addon are transparent to the object.

An addon can add attributes, if it does not duplicate names. Usually, the addon will contain the identity of the relation object it is tied to. In case of an $1-n$ relation, this attribute is a set. The addon is only added the first time an object engages in a relation. When the object engages in more relations of the same type, extension with an addon only means that an element is added to this set.

Methods can only be added. There is no mechanism to modify the behaviour of existing methods, other than specifying a rule on an existing method. Rules are treated the same as methods with regard to specialisation.

The addon mechanism offers a number of advantages over using inheritance to specialise objects. The key to these advantages is the observation that object specialisation is related to the role of an object. An object is specialised in order to play a role. In an inheritance hierarchy we would need a separate class for each possible combination of object extensions. Clearly, this leads to a combinatorial explosion of the number of classes in the hierarchy [17]. In DEGAS, this observation has lead to the extension of an object with an addon, when it engages in a relation. The addon defines the role the object plays in the relation. It gains methods to deal with the relation. Rules specify what information must be passed to the relation, while lifecycles define the access of the relation to the methods of the object.

A number of other approaches are based on this observation. For example, Aspects by Richardson and Schwarz [20] are also dynamic extensions to objects. There is no link between aspects and relations. Although aspects can have aspects themselves, there is no possibility of interaction between aspects of the same object. This means that interaction between relations of an object, or multiple roles, in the way shown in our example is not possible using aspects. A database programming language offering an extensive role mechanism is Fibonacci [4]. Its object specialisation mechanism is more complex than the DEGAS addon mechanism. For example, it has multiple inheritance between roles. This is caused by the strongly typed functional nature of Fibonacci. In DEGAS multiple inheritance is not needed, since addons need no information about other addons.

There is no treatment of rules or time in both Aspects and Fibonacci.

Another extension of an object oriented language with roles is given by Gottlob in [15]. Here, an implementation of an object specialisation mechanism in Smalltalk is given. A number of characteristics of roles are given by Gottlob:

- Various roles of an entity may share common structure and behaviour.
- Entities can acquire and abandon roles dynamically.
- Roles can be acquired and abandoned independently of each other.
- Entities exhibit role-specific behaviour.
- Roles restrict access to a particular context.
- Entities may occur repeatedly in the same type of role.

These characteristics also apply to DEGAS addons except for the role-specific behavior. Gottlob allows roles to redefine methods of the object they are extending. This is not allowed in DEGAS. Gottlob's approach, however, does not take the link between roles and relations into account.

An extensive conceptual study and formalisation of objects with roles can be found in [27]. Here it is observed that there are static classes, dynamic classes and roles. Objects cannot migrate between static classes. Hence, these are equivalent to the classes in DEGAS. Dynamic classes are based on dynamic partitions of a static object class. Objects can migrate between dynamic classes, although this may be subject to lifecycles. Roles are dynamic classes that do not partition an object class. In addition an object can play multiple roles. In DEGAS the latter two are both modelled using addons. Dynamic class migration is specified in the lifecycle of an object. Migration is achieved through the gain and loss of addons. Roles are tied to relations. When engaging in a relation an object will gain the addon that specifies its role in the relation. The main difference is that DEGAS only distinguishes between inherent and transient capabilities of an object.

Lifecycle specification in addons

The semantics of object specialisation through addons in DEGAS is relatively straightforward, if we consider attributes, methods, and rules. These are simply added to the capabilities already present. The combination of the lifecycles of an object and an extending addon is more complicated. This is due to two potentially conflicting requirements on lifecycle specification by addons.

The main requirement is that the lifecycle of the extended object must conform to the original object. In other words, the lifecycles specified in an addon must not violate the lifecycle of the original object. To put it in process algebraic terms using the abstraction operator, we require:

$$\partial_H(C_A) = C_O$$

where C_O is the lifecycle of the original object O, C_A the lifecycle of O extended with addon A, and H is the set of methods defined in the addon. The effect of the abstraction operator ∂ is to filter away the actions in the set that we abstract from. For example,

$$\partial_{\{Y,Z\}}(A;B;Y;C;Y;Z) = A;B;C$$

A simple strategy to satisfy this requirement is to put the lifecycle of the addon in parallel with the original lifecycle and only allow methods of the addon itself to appear in the addon's lifecycle. An example of this approach is found in MOKUM [21]. This, however, is a severe restriction, since it prevents the redefinition of the original lifecycle of an object. This redefinition is necessary, if we wish to intersperse the actions of the addon with those of the original object. An example of this is constraint checking in a graphical database, as described in [3].

The solution chosen in DEGAS allows redefinition, while respecting the original lifecycle of an object, by using communication merge as a combination operator with a communication function

that merges identical actions, i.e., $\gamma(\alpha, \alpha) = \alpha$, as defined in the previous section. To show that we thus satisfy the first requirement, suppose object O has the lifecycle:

$$A; B; C$$

If we specify in addon A the lifecycle:

$$A; X; B; Y; C$$

then the resulting lifecycle for the extended object will be:

$$(A; B; C)|(A; X; B; Y; C) = A; X; B; Y; C$$

Thus, we can extend the lifecycle of an object with methods from an addon.

Communication merge also helps to avoid conflicts of lifecycle redefinition in different addons to an object. To illustrate this, consider the following situation, where two addons try to modify an object's lifecycle. An example are multiple constraints added to objects in a graphical database, as shown in [3]. The original object O has the lifecycle:

$$A; B$$

Object O engages in a relation that demands that O must execute action C between A and B. Thus, the addon A_1 requires O to follow:

$$A; C; B$$

A second addon A_2 desires an action D to be inserted in the lifecycle of O:

$$A; D; B$$

With communication merge, the result is that the extended object O_x follows:

$$A; (C \| D); B$$

Since O abstracts from C and D, A_1 from D, and A_2 from C, this lifecycle satisfies all defined lifecycles.

7. DEGAS IN A BROADER CONTEXT

In this section we show DEGAS in a broader context than active databases. Not only is the DEGAS notion of object autonomy the natural consequence of the integration of rules in an object database, it also supports currently foreseen developments in computing, networking and integration of information systems. These contribute to the need for systems built of autonomous components. Autonomy in this case implies extreme distribution. A more elaborate motivation for object autonomy can be found in [2].

Developments in Technology

Extreme distribution is motivated by a number of developments foreseen in computer systems in the nearby future. These are the emergence of massively parallel computer systems and the coupling of existing computer systems over networks. These have in common that any form of central control will pose a large amount of overhead on the system. In a massively parallel computer centralisation of decisions, for example regarding resource allocation or invocation of active rules, poses overhead on the system. Enough overhead to make it a considerable factor in the performance of such a system.

Similar problems are posed by the possibility of information systems running on networks of mobile computers, in its ultimate form known as ubiquitous computing. A lot of effort has been put into making databases interoperable over a network. Still, it will be very difficult to come up with a scheme that can keep up with the sheer size of such a network and the speed of changes in the network caused by its mobile character. It seems a better idea to build an inherent flexibility into the components, such that they can function with as little global information as possible.

Because of the problems of central control in these environments, control must be distributed to components of the system. In other words, the components are forced to be autonomous.

Integration of Information Systems

There is a strong trend to increasing integration of systems in chain information systems or through a public information infrastructure. Such systems merge (parts of) information systems of various owners into one big information system. Examples are integrated information systems for suppliers and customers and the trading system at the stock exchange, as shown in this article. However, nobody wants to give up control over his part of such a system. In addition one organisation may want to integrate its information system with a number of inter-organisation systems. An example of this is a supplier of tyres, who sells these to several car manufacturers, that all have an information system for their own chain of suppliers and resellers.

An inter-organisation information system is made up of parts that are not subject to any form of central control. This means that we need information systems that function without central control of the components. In addition, different parts of an organisation's information system may be exported to different inter-organisation systems. This means that access control can differ at a very fine grain in the system. For each component we want to be able to define who has access to what.

These developments again force components of a system to function without central control. In addition they point at a need to be able to define access control in a system at a very fine grain. Autonomy of components makes this possible.

Autonomy is the Solution

All the developments mentioned above foster a need for systems composed of autonomous components. The difficulties with central control of a system can be overcome by distributing control to parts of the system, or by building inherent flexibility into the parts of the system. The result will be autonomy for the components of such a system. We also signalled a development towards the sharing of data with outsiders. Approaching data from multiple sources as one database while the owners retain control, means autonomy for the components. Exporting data to multiple inter-organisation information systems at a time, asks for an inherent flexibility that autonomous components can offer.

DEGAS offers a formal model to support the development of systems of autonomous components. This is achieved by basing the DEGAS model on autonomous objects. We have chosen object as the level of autonomy, because of its obvious advantages in modelling an information system. Object autonomy also has the advantage of generality, because the complexity of the objects may be arbitrary. This means that the model can also be used for autonomous components at a higher abstraction level, as long as its behaviour can be described in DEGAS.

Autonomous objects and Agents

In recent years the notion of agents has received considerable attention as a paradigm for software development. Research has focussed either on the specification of agents by logic, see for example [22], or on the implementation of special purpose agents, for example to schedule meetings (see [1]). Since the logics used to specify agents are relatively complex, there is a gap between these two approaches. To bridge this gap we need simple general purpose agents. Autonomous objects are a first step towards this kind of agents.

Another area where agents can be useful, is the design and analysis of information systems. For example, Yu et al [28] propose an agent-oriented framework for the specification of information systems. This framework consists of two parts, one, Albert, to specify agents in an information system and the other, **i***, to understand and redesign the organisational context of the information system. If we want to apply this framework also to the design and implementation phase of information system development, we need a database programming language that supports the modelling notions used in the specification phase. Autonomous objects in DEGAS offer such support through their rule and lifecycle specifications.

8. CONCLUSION

In this paper we introduced DEGAS, an active temporal data model, using an application with a highly dynamic content, the stock market, as an example. The relation and addon mechanism of DEGAS, where capabilities are only present when they are needed makes DEGAS especially useful for this kind of applications. In addition addons and relations offer a clean mechanism to implement roles.

DEGAS emphasises the integration of the dynamic and static parts of an application. This integration is achieved through the complete encapsulation of an object's behaviour. This contributes to the autonomy of objects, an important factor in the construction of highly distributed information systems.

An important contribution in the field of active databases is the temporal element of DEGAS. The notion that the state of an object is formed by its complete history makes it possible to achieve temporal database functionality in DEGAS. The benefits of the inclusion of the object history in an active database are found in a direct definition of the semantics of rules and lifecycles in terms of process algebra.

Further research on DEGAS is concerned with a query model for DEGAS. We are focussing on the consequences of object autonomy for query processing and on the use of the addon mechanism to help counter impedance mismatch between query and programming language. A prototype DEGAS object kernel has been implemented with the execution cycle shown in this paper as the central element.

Acknowledgements — This research is supported by SION, the Foundation for Computer Science Research in the Netherlands through Project no. 612-323-424.

REFERENCES

[1] Special issue on intelligent agents. *Communications of the ACM*, **37**(7) (1994).

[2] J.F.P. van den Akker and A.P.J.M. Siebes. A data model for autonomous objects. Technical Report CS-R9539, CWI, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, Available through WWW (`http://www.cwi.nl/~vdakker/`) (1995).

[3] Johan van den Akker and Arno Siebes. Applying an advanced data model to graphic constraint handling. In *Proceedings of the 5th Eurographics Workshop on Programming Paradigms in Graphics*, pp. 1-16, Maastricht, The Netherlands (1995).

[4] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In Rakesh Agrawal, Sean Baker, and David Bell, editors, *Proc. of the 19th Intl. Conf. on Very Large Data Bases (VLDB)*, Dublin, Ireland (1993).

[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] Herman Balsters and Maarten M. Fokkinga. Subtyping can have a simple semantics. *Theoretical Computer Science*, **87**:81-96 (1991).

[7] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Proceedings of the International Symposium on the Semantics of Data Types*, pp. 51-68, Berlin, Germany. Springer (1984).

[8] Stefano Ceri et al. *Active Rule Management in Chimera*, chapter 6 in [26] (1996).

[9] U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, **17**(1):51–70 (1988).

[10] Klaus R. Dittrich and Stella Gatziu. Time issues in active database systems. In *Proc. of the Intl. Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, USA (1993).

[11] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. The active database management system manifesto: A rulebase of ADBMS features. In T. Sellis, editor, *Rules in Databases: Proc. of the 2nd International Workshop*, pp. 3-17, Athens, Greece. Springer (1995).

[12] Avigdor Gal and Arie Etzion, Opher abd Segev. TALE: A temporal active language and execution model. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Advanced Informations Systems Engineering - Proc. of CAiSE'96*, pp. 60-81, Heraklion, Crete, Greece. Springer, LNCS 1080 (1996).

[13] 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*, pp. 399-415. Morgan Kaufmann (1991).

[14] Seymour Ginsburg. *Object and Spreadsheet Histories*, chapter 12 in [25].

[15] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, **14**(3):268-296 (1996).

[16] Eric N. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, **8**(1) (1996).

[17] David McAllester and Ramin Zabih. Boolean classes. In M. Meyrowitz, editor, *Proceedings OOPSLA'86*, pp. 417-423 (1986).

[18] L. Edwin McKenzie Jr. and Richard T. Snodgrass. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Computing Surveys*, **23**(4):501-543 (1991).

[19] 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).

[20] Joel Richardson and Peter Schwarz. Aspects: Extending objects to support multiple, independent roles. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 298-307 (1991).

[21] R.P. van de Riet. MOKUM: An object-oriented active knowledge base system. *Data and Knowledge Engineering*, **4**:21-42 (1989).

[22] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, **60**:51-92 (1993).

[23] A.P.J.M. Siebes, J.F.P. van den Akker, and M.H. van der Voort. (un)decidability results for trigger design theories. Technical Report CS-R9556, CWI, Amsterdam, The Netherlands, Available through WWW (`http://www.cwi.nl/~vdakker/`). (1995).

[24] A. Prasad Sistla and Ouri Wolfson. Temporal conditions and integrity constraints in active databases. In *Proc. of the 1995 SIGMOD International Conference on the Management of Data*, pp. 269-280, San Jose, CA, USA (1995).

[25] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, Redwood City, CA, USA (1993).

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

[27] 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).

[28] Eric Yu, Philippe Du Bois, Eric Dubois, and John Mylopoulos. From organization models to system requirements: A "cooperating agents" approach. In *Proc. of the Third International Conference on Cooperative Information Systems (CoopIS'95)*, Wien, Austria (1995).