DEGAS: Capturing dynamics in objects

Johan van den Akker* and Arno Siebes

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

Abstract. In this paper we introduce DEGAS (Dynamic Entities Get Autonomous Status), an active temporal data model based on autonomous objects. 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. 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.

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 [21, 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. To achieve an effective unification we also incorporate elements of a third area,

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

temporal databases. 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 First, we introduce the main concepts of the DEGAS model. Next, we show the use of these concepts by modelling an example. Then, we discuss the DEGAS model on a more formal level. After that we look at the broader context, motivating the development of DEGAS. We conclude with issues for future research.

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 [15].

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. 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 engages 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 can be 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.

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 only extends the Marketmaker with the SupplyDemandAddon, that 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 should place 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 will use the information from the newspaper to influence decisions about his shares. This is specified in the person object as follows:

```
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
```

In the person and marketmaker objects the methods define that the object engages in relations. 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. 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, guards are used to restrict access to its methods.

```
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
```

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. 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.

```
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
```

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 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. First, we give the specification of the InformedPerson addon, that extends a person who has a subscription to a newspaper.

```
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)($t_1$);goodNews(company)($t_2$)
    if $t_2 - t_1 \leq 7$ days
    do tryToBuy(company,transactionPrice)
EndAddon
```

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 following specification 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.

```
Addon InformedOwner
Extends InformedPerson,Shareholder
Attributes
    Key : $\mathcal{P}\langle$ Subscription : Oid, Share : Oid $\rangle$
Lifecycles
    ExtendInformedOwner*
    DropInformedOwner*
Rules
    On badNews(company)($t_1$);badNews(company)($t_2$)
    if ($t_2 - t_1$) $\leq$ 7 days && transactionPrice($t_2$) $\leq$ transactionPrice($t_1$)
    do tryToSell(transactionPrice)
    On goodNews($t_1$);badNews($t_2$)
    if $t_2 - t_1 \leq$ 7 days && transactionPrice($t_1$) == max(transactionPrice, $t_1$, $t_2$)
    do tryToSell(transactionPrice)
    On DropShareHolder do DropInformedOwner
    On DropSubscription do DropInformedOwner
EndAddon
```

The diagram in Figure 1 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 Discussion of DEGAS concepts

In this section we discuss the different elements of the DEGAS model in more detail. First, we will look at the different parts of a DEGAS object specification. After that, we will take a closer look at relations and addons. For a full formal description of DEGAS, including its semantics, the reader is referred to [3].

Attributes and Methods Attributes and methods are straightforward in DEGAS. Attributes in an 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]. In DEGAS method definition allows assignment to attributes. In addition, we can map a method over a set. A method can also call other methods.

Lifecycles A method call can be executed on an object, if it follows one of the lifecycles specified. The semantics of lifecycles follows process algebra [5]. If multiple lifecycles are specified, then these are executed in parallel. Suppose we have defined the following set of lifecycles on an object O.

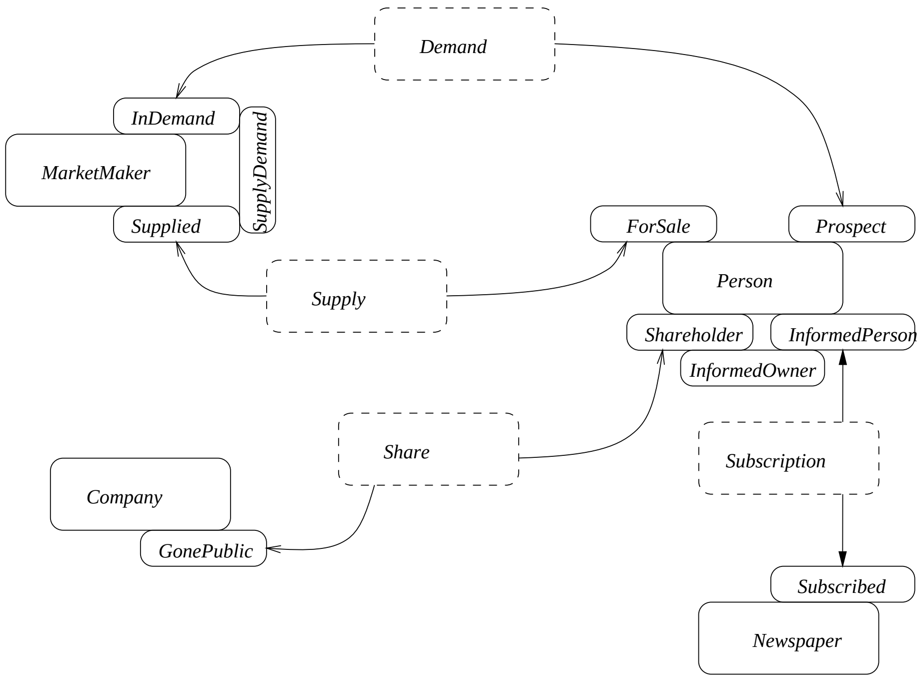

Fig. 1. The Degas model for a financial market

Lifecycles
C_1
C_2
$\vdots$
C_n

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

$$C_1 \| C_2 \| \ldots \| C_n$$

This means that the lifecycles specified are executed in parallel. Lifecycles can be checked using finite automata. This follows from the fact that lifecycles are regular expressions. In an object we have a finite automaton associated with every lifecycle. 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 History of an Object The execution of a method modifies the state of an object. In Degas, an object's state is its complete history, which is represented as a sequence of snapshot states. In temporal database research [20], the term snapshot state denotes the state of an object at a point in time when we abstract from the temporal dimension [14]. In DEGAS a snapshot state contains a timepoint, the attributes and their valuation at that time, and the method call that brought the object in the state. As an example, the following is a piece of the history of a share object:

$$(13:00:00, \langle currentPrice : real, person : oid, company : oid, \rangle,$$
$$\langle currentPrice = 54.25, person = Johan, company = Philips \rangle,$$
$$transferOwnership(Johan, 54.25))$$

$$(13:02:00, \langle currentPrice : real, person : oid, company : oid \rangle,$$
$$\langle currentPrice = 55.25, person = Arno, company = Philips \rangle,$$
$$transferOwnership(Arno, 55.25))$$

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. Type information is included because of the dynamic nature of the capabilities of DEGAS objects, which can be changed through addons.

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 history is the central element in the formalisation of DEGAS. Method definitions specify the state transitions possible in DEGAS. The sequence of method calls is restricted by the lifecycles of an object. In addition rule triggering is described in terms of the state history.

Although the natural combination of temporal and active databases has been suggested by different authors [10, 21], there are no active data models that incorporate the history of a database. Work on the temporal specification of rules has been done by Sistla and Wolfson [19]. This approach is based on temporal logic. It focusses only on the condition of the rules, which means that there is no general history mechanism in their approach.

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 $\langle Event \rangle$ **if** $\langle Condition \rangle$ **do** $\langle Action \rangle$

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.

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 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 2.

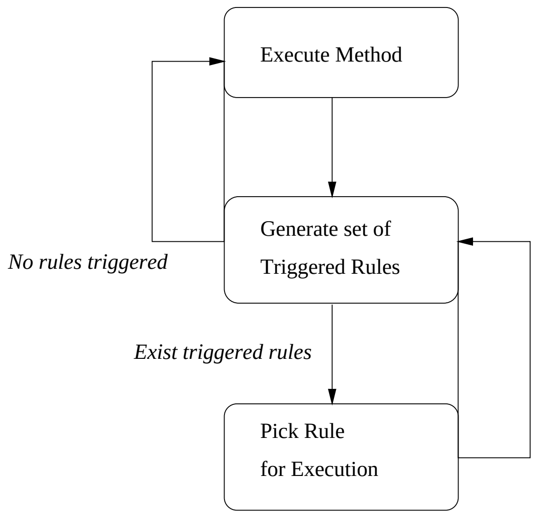

Fig. 2. The execution of an autonomous object

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 not picked are dropped. During execution of a rule's action, a new set of triggered rules is constructed.

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 ¬ 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;¬ 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 [18]. 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 [12] 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 DEGAS this kind of manipulation is offered through the addon mechanism.

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 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.

In an addon, lifecycles can be added to the set of existing lifecycles. As an example, suppose we have the following lifecycle definition in object O:

Lifecycles
$(A;B)^*$
$(C;D)^*$

Clearly, O follows the process:

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

If O is extended with addon A with the following lifecycle definition:

Lifecycles
$(K;L)^*$

The specialised object O will follow the process:

$$(A;B)^* \| (C;D)^* \| (K;L)^*$$

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 tied 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 [13]. 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 [16] 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.

An extensive conceptual study and formalisation of objects with roles can be found in [22]. 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.

5 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. 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 [17], 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 [23] 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.

6 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. We elaborate on the temporal aspects of DEGAS in a forhtcoming paper.

A prototype implementation is underway. The simplicity of the DEGAS rule model is expected to facilitate a performant system. To complement the data model a query model will be formalised for DEGAS. Basically, a query is the specification of a set of objects. Further research will focus on the integration of temporal and active databases.

References

1. Special issue on intelligent agents. *Communications of the ACM*, 37(7), July 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, 1995. Available through WWW (`http://www.cwi.nl/~vdakker/Publications.html`).
3. 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/Publications.html`).
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*, pages 51-68, Berlin, Germany, 1984. Springer.
8. Stefano Ceri and Rainer Manthey. Consolidated specification of Chimera (CM and CL). Technical Report IDEA.DE.2P.006.01, IDEA, ESPRIT Project 6333, 1993. Available by FTP from rodin.inria.fr:/pub/IDEA/DE.2P.006.ps.gz.
9. U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.
10. Klaus R. Dittrich and Stella Gatziu. Time issues in active database systems. In N. Pissinou, R.T. Snodgrass, and R. Elmasri, editors, *Towards an Infrastructure for Temporal Databases: report of an international ARPA/NSF workshop*, Tucson, AZ, USA, 1994. University of Arizona, Dept of Computer Science, TR 94/01.
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*, pages 3-17, Berlin, Germany, 1995. Springer.
12. 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, San Mateo,. CA, USA, August 1991. Morgan Kaufmann.
13. David McAllester and Ramin Zabih. Boolean classes. In M. Meyrowitz, editor, *Proceedings OOPSLA'86*, pages 417-423, 1986.
14. 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, December 1991.

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. 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*, pages 298-307, 1991.
17. Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51-92, 1993.
18. 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, 1995. Available through WWW (`http://www.cwi.nl/~vdakker/Publications.html`).
19. 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*, pages 269-280, San Jose, CA, USA, 1995.
20. 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.
21. Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, CA, USA, 1995.
22. 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.
23. 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, May 1995.