Centrum voor Wiskunde en Informatica

**REPORT** *RAPPORT*

DEGAS: A temporal active data model based on object autonomy

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

Computer Science/Department of Algorithmics and Architecture

**CS-R9608 1996**

Report CS-R9608
ISSN 0169-118X

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

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

# DEGAS: A Temporal Active Data Model based on Object Autonomy

Johan van den Akker
Arno Siebes

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

**Abstract**

This report defines DEGAS, an advanced active data model that is novel in two ways. The first innovation is object autonomy, an extreme form of distributed control. In comparison to more traditional approaches, autonomous objects also encapsulate rule definitions to make them active. The second innovation of DEGAS is its temporal aspect. Active databases have an inherent temporal element in the specification and detection of event patterns that trigger rules.

Autonomous objects, the foundation of the DEGAS data model, are independent processes. Their definition includes their complete behaviour, both *potential* behaviour in the form of methods and lifecycles and *actual* behaviour in the form of active rules. Relations between objects are objectified. The specialisation mechanism provided by DEGAS is a clean addon mechanism well suited to model dynamic evolution of objects in conjunction with relations. In this report we give a full syntactic and semantic definition of the data model. The state of an autonomous object includes its complete history. This allows the active behaviour of the object to be defined in a purely local way.

*CR Subject Classification (1991):* H.2.1 [Information Systems]: Logical Design - *data models*, H.2.3 [Information Systems]: Languages - *database programming languages.*

*Keywords and Phrases:* database programming languages, active databases, time, autonomous objects, rules, data model

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

## Introduction

Recent years have shown an increasing move towards distribution of data in informations systems. This move is triggered by a number of simultaneous developments in technology and business. These lead to the emergence of distributed information systems where centralised control is either unfeasible, because of a large number of mobile nodes, or undesirable, because data ownership is with different organisations. Hence, there is a need for systems with components that function autonomously. This kind of distributed information systems needs database support. The DEGAS[1] data model proposed in this report offers a database framework for such distributed information systems.

In DEGAS, the unit of autonomy is the object. To function autonomously, it must encapsulate every aspect of its behaviour. DEGAS integrates two approaches to add behaviour to databases, namely object-oriented databases and active databases. Furthermore, we recognise that a dynamic view of a database should include the history of the data, thus incorporating elements from temporal database research as well.

**Roadmap** The first section gives a motivation and a definition of object autonomy. The second section discusses th relation between temporal and active databases. After that, Section 3 introduces the main concepts of the DEGAS data model. These are illustrated by an example in Section 4. Section 5 then gives the syntax of DEGAS. The basic semantics of DEGAS is then defined in Section 6. In Section 7 we discuss communication between DEGAS objects. Section 8 extends the semantics with time. Section 9 then discusses relations between objects. Section 10 defines the execution of a DEGAS object and, consequently, the constraints on the history of an object. Section 11 then discusses a number of decisions we made during the design of the data model. We discuss some related work in Section 12.

# 1 Object Autonomy

The key notion of the DEGAS data model is object autonomy. Autonomy means two things. First, all static and dynamic aspects of an object are encapsulated with the object itself. Second, the object is subject to its own, local, control only. In DEGAS, there is no central control. Hence, we take distribution to its extremes by viewing each object as a process.

The motivation for the introduction of object autonomy into a database scene is twofold. On one hand it is a natural further development of the current state of affairs in active databases. On the other hand it provides a flexible testbed for distributed information systems.

### 1.1 From Activity to Autonomy

An active database [36, 42, 17] is a database management system that incorporates production rules or triggers. Rules are usually formulated as an Event-Condition-Action (ECA) triple. The action is executed on the specified event, if the specified condition holds. The most obvious use of these is to offer flexibility in the enforcement of integrity constraints. However,

[1]The acronym DEGAS stands for Dynamic Entities Get Autonomous Status. It also follows the popular tradition of naming database systems after French painters.

much wider use can be found for rules in databases. In fact it is possible to encode the entire dynamics of an information system as rules in an active database system.

Making an object database active means that we will have to consider the place of rules in the object model. Rules can be integrated with the database by making them objects as well [14, 15]. This approach puts emphasis on easy manipulation of the rules. However, the most important advantage of object-orientation is the encapsulation of data and behaviour in one object. Rules are part of the behaviour of objects. They describe what actions are triggered in specific situations.

In DEGAS we take encapsulation to its logical consequence. *All* behaviour is encapsulated with an object. Thus, the object is independent of other parts of the system. In other words, an object definition is *autonomous.*

Other examples of object-oriented DBMSs that offer encapsulation of rules in objects are SAMOS [19] and Chimera [11]. However, these systems offer a hybrid model. In both systems it is still possible to define rules separate from a class. Thus, we do not have a single place to look for the behaviour of an object.

## 1.2 Motivation for Autonomy

### 1.2.1 Developments in Technology

As we stated above autonomy is in part about doing away with central control. The motivation for this can be found in a number of developments foreseen in computer systems in the nearby future. These developments trigger the emergence of information systems based on large mobile networks. In such systems, central control will imply a large amount of overhead.

The development of powerful mobile computers and the spread of wireless communications makes large networks of mobile computers possible. A broad category of information systems will rely on such networks. As examples we have companies with a large number of sales representatives or organisations managing a fleet of ships or aircraft. Each mobile node contains data. Ideally, we want to access all data in such a network as one large database. Although a lot of effort has been put into making databases to interoperate, it will be very difficult to devise a scheme clever and flexible enough to keep up with the size and volatility of such a network.

A better approach is to have inherent flexibility built into such a system. In this situation an almost arbitrary collection of objects is a functioning database. The collection of objects making up the database may vary over time, but there is no system "authority" that needs to keep up with their changes. In such a system an object is autonomous, because it functions largely independent of other elements of the system.

### 1.2.2 Developments in Business

Outside the area of computing there are a number of other developments that promote autonomy of system components. Although there is a movement to increase integration of systems in chain information systems or through a public information infrastructure, nobody wishes to give up control of his part of such a system.

**Inter-Organisation Information Systems** The developments in networking has a number of effects in the way organisations interact. For example, it enables a manufacturer to integrate his information system with those of his suppliers. However, when an organisation couples its information system with other organisations, it will not wish to give up control over its own system. In particular everyone wishes to determine the information seen by outsiders. Although the components are under control of separate owners, we do want to approach such a chain information system as a monolithic information system.

The data in the chain information system and the individual information systems overlap. In fact the chain information system is made up from subsets of the data in each corporate information system.

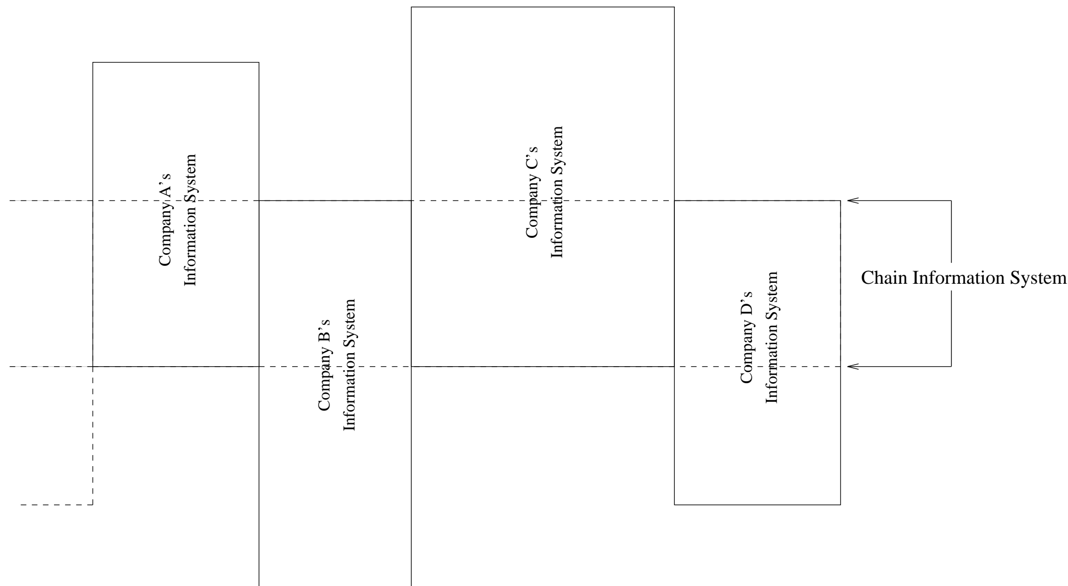


Figure 1: The integration of ISs in an inter-organisation IS

Different collections of objects make up the corporate information systems and the chain information system. It is important to note here that a company will probably be involved in multiple value chains of suppliers and customers at the same time. This means that it has to export data to the information system of each value chain.

The problem related to mobile computing mentioned in the previous subsection, resurfaces again here in another guise. Here, we do not have a set of objects evolving over time, but different sets of objects making up different databases simultaneously. The choice is again between devising a clever scheme for a chain information system that gets data from different composing databases, and building in inherent flexibility at the object level. This flexibility should allow an organisation to decide per object to make it visible to partners in a chain information system. Thus, making objects autonomous facilitates integration of an organisation's data into multiple information systems at a time.

Another example of an inter-organisation information system is the trading system of the stock exchange. Every party in the market would like to approach the computerised market

as a whole in order to obtain information. However, a sensible company would not hand over any control of their computerised trading system to third parties. It would also want to have complete control over the flow of its data to other parts of the system. Clearly, such a system consists of autonomous components.

**Business Modelling** The introduction of information systems in an organisation is nowadays often used to reevaluate the way business is conducted by the organisation. This activity has risen to fame in the last years under the name Business Process Redesign. Because of the focus on what an organisation does, business modelling as part of the analysis phase of information systems development has focussed on the dynamic aspects of an organisation.

Therefore dynamic modelling [35, 43] has received a considerable amount of attention. Most dynamic models feature actors that manipulate data according to scripts or scenarios. The reactions of actors to events are described by rules. These actors are clearly active and independent of other actors. A data model with facilities to implement such autonomous actors integrated with other data objects would offer a good model to use in the next phase of the development of an information system.

## 1.3 Autonomy is the Solution

All these developments outlined above lead one to conclude that there is a need for systems composed of autonomous components. In Section 1.1 we showed that object autonomy is the ultimate consequence of incorporating active rules in an object oriented database. Section 1.2.1 pointed out a number of developments in computing that make central control of a system very difficult. These difficulties 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.

In Section 1.2.2 we indicated 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. It also showed that current developments in the modelling of organisations tend to emphasise the active and autonomous behaviour of different components of an information system.

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 the 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. For example, an active class of passive objects, where there is activity at the class level but not at the objects, also fits in this model.

## 1.4 Defining Autonomy

The notion of autonomy is also encountered in the area of distributed and federated database systems [31]. This area studies inter-operation of multiple database systems either as one,

distributed, database system, or as a looser federation of databases. These systems vary in the degree of freedom of the component databases, hence several authors have formulated criteria of autonomy to distinguish the various systems.

Sheth and Larson [31] distinguish four kinds of autonomy for a component database of a database federation. These are:

**Design autonomy** The freedom to choose the design of any part of the database, from semantic interpretation of the data to the implementation of the database.

**Communication autonomy** The freedom to decide when and how to communicate with other component databases.

**Execution autonomy** The freedom to decide when and how to execute operations on the database.

**Association autonomy** The freedom to decide whether and how much functionality the component database shares with the federation.

Other criteria for the autonomy of the nodes in a distributed database system were formulated by Garcia-Molina and Kogan [18]. They distinguish the following facets of autonomy:

**Heterogeneity** A component database may choose its own way to manage data and transactions.

**Naming Autonomy** The freedom a component has in choosing its names.

**Setting priorities for foreign requests** The freedom to decide whether and how a request from outside is processed.

**Transaction control autonomy** The freedom a component has in transaction management, i.e. scheduling, locking, and aborts.

The criteria of Sheth and Larson cover a wider area than those of Garcia-Molina and Kogan. For example, heterogeneity and naming autonomy are included in the single category design autonomy. Setting priorities for foreign requests and transaction control autonomy are included in the criteria communication autonomy and execution autonomy, respectively.

We should note that the motivation for autonomy for component systems is partly similar to the motivation for object autonomy given earlier in this section. Garcia-Molina and Kogan [18] state the following arguments in favour of node autonomy in a distributed database: organisational issues, diversity of local needs, data security, lower costs and containment of failures and bugs. These arguments also apply to object autonomy. For example, containment of failures and bugs means that it is desirable, if the failure of one object affects other objects as little possible.

The applicability of autonomy criteria for component systems to object autonomy is limited in some respects. The aim of DEGAS is to study systems of autonomous componenents of various scales, from objects upward. Hence, not all criteria for autonomy of component databases are useful for the definition of object autonomy. Design autonomy for an object would mean that we would only define a communication interface between objects. In DEGAS, however, we are

also concerned with the internals of an object. Communication autonomy for an object means that it can decide for itself if and how to answer a message. In other words, the sender of a message has no guarantees about the reaction of the receiver. Execution autonomy implies that an object decides internally what actions to execute and when to execute them. Finally, the association autonomy of an object means that an object has the freedom to decide what parts of itself it makes visible to other objects.

In the discussion of DEGAS in this report the term autonomy refers to the following criteria. We relate these criteria to the criteria outlined by Sheth and Larson:

**Complete encapsulation of the behaviour of an object** This means that every aspect of an object's behaviour is specified on the object itself. Hence, the behaviour of an object, given certain stimuli from outside, is determined locally. This is a relatively weak form of design autonomy.

**Strictly regulated access to an object** A DEGAS object will specify exactly what objects have access to its actions. Relations between objects specify exactly what data is shared. This criterium corresponds to association autonomy.

**Minimal guarantees about an object's behaviour to other objects** A DEGAS object guarantees as little as possible on its behaviour to other objects. If it gives guarantees, these are specified explicitly. This corresponds to communication and execution autonomy.

**Minimal dependency of an object on the behaviour of other objects** A DEGAS object assumes as little as possible about the behaviour of other objects. If it makes an assumption, it has to verify this assumption explicitly. This is another aspect of communication and execution autonomy.

A further guiding principle is that if an object gives up autonomy, this must be explicitly specified.

Not surprisingly, the degree of autonomy of a DEGAS object is weaker than the autonomy of a component database in a database federation. The smaller granularity of the autonomous component is the main reason for this. Autonomy in a certain matter means that the functionality that goes with it, must be carried locally. Clearly, an object can carry less functionality than a complete component database system.

## 2 Time in Databases

### 2.1 Time in Active Databases

The key feature of active databases [36, 42, 17] are production rules. Usually, these are defined as event-condition-action (ECA) triples. The event specification may be a complex event expression composed of multiple basic events, such as method calls [14, 19] and time events [13, 21, 19]. Since rule definitions specify sequences of events over time, an active database has an inherent temporal element, as observed by Dittrich [16] and Ceri and Widom [42].

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 some information about the order in which events occurred.

This inherent temporal element in active databases raises 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 offered.

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. This way a rule can be triggered on method calls with certain values for the parameters only. 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 part of the 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 timestamp. 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. Therefore, the state of a DEGAS object includes its history, i.e. a record of past states and method calls.

## 2.2 Temporal Databases

An information system reflects the state of a part of the real world, which is subject to continuous change. In many applications we need facilities to consult data from the past. For example, a bank will want insight in the amount of money feeding your bank account during the past year to assess your credibility. This need to store data in relation to time is supported by temporal databases. In particular, historical databases record past states of the database.

Temporal databases [40] are founded on data models that include a temporal dimension. Not surprisingly most work on temporal databases is based on an extended relational model. If a clock is available, adding timestamps to data is relatively straightforward in this case. For example, to each tuple we add an attribute that indicates when this tuple was valid. Another approach is to record the time of validity per attribute. An overview of temporal relational algebras and the design decisions in defining them is given by McKenzie and Snodgrass in [25].

Object-based temporal data models are less common. In [45] it is shown how a temporal dimension can be brought into the OODAPLEX model. In this model, every function application to an object is parameterised with time in order to get the object state at that time. Another model that stores past object states is Ginsburg's object history formalism [20]. Here, the state of an object is a sequence of past states representing the history of the object. Again, we can also record temporal data on an attribute basis. This is proposed as a special case of versioning by Sciore [30].

The complexity of the temporal dimension in a database can differ widely. A choice must be made between real and discrete time. In addition there may be two temporal dimensions: valid time, to indicate when a value held in the real world, and transaction time, to indicate when a value was entered into the database. The simplest temporal databases are the historical databases we encountered above. These usually only incorporate transaction time to record the time data was entered into the database.

Querying a temporal database is more complex than querying a database without a temporal dimension. Not only can we ask for tuples or objects, we can also query the database to get time intervals or points in time. A simple extension of snapshot query languages is the parameterisation of queries with time. This way we can query the database state relative to time. More sophisticated temporal query languages, such as TSQL2 [34], offer the possibility to query for time intervals. For example, we can ask for what time intervals a condition holds or did hold. Other operators on time are defined in order to manipulate time intervals, such as an operator to get the overlap of two time intervals.

In the DEGAS model we only incorporate a historical record of an object's state. Hence, DEGAS is not a full temporal database. It uses transaction time only and does not allow rewriting history. This choice was made for two reasons. First, in most applications a historical database is sufficient. Second, the history of a database is the part of a temporal database that is of interest for the integration of active and temporal databases.

# 3 Concepts

This section discusses the concepts underlying the DEGAS model, introduced in the previous section. The foundation is formed by autonomous objects. After that, we will look at the relations between individual objects. This results in a structure for a database of autonomous objects.

## 3.1 An Autonomous Object

The basic idea behind an autonomous object is that we wish to encapsulate all aspects of an object. The behaviour of an object is formed by methods, rules and lifecycles. Together with the attributes these three components form the *capabilities* of an object.

Methods define the *possible* actions on the data encapsulated in the object. Lifecycles define what an object is *allowed* to do. Together methods and lifecycles form the *potential* behaviour of an object. Rules state actual actions to be taken in certain situations described in terms of events and object states. Thus, rules define *actual* behaviour of an object.

The inclusion of rules in an autonomous object is an extension of more traditional object-oriented databases. These only specify potential behaviour in the form of methods and life-cycles. The DEGAS model also specifies the actual behaviour of an object as far as possible in the object itself by including rules in an object definition.

An autonomous object in DEGAS belongs to a class of objects. The definition of an object class has the usual specification of attributes and methods plus the following behavioral parts:

**Rules** The active behaviour of an object is specified in terms of ECA rules. The event specification is a process algebraic (ACP [7]) expression that refers to the local event history of the object.

**Lifecycles** The behaviour of an object is restricted by lifecycles. Like event expressions these are specified in the form of basic process algebraic expressions.

Objects can be specialised through the addon mechanism. An addon defines attributes, methods, rules and lifecycles. If an object is extended through an addon, it gains these capabilities. These cannot be distinguished from the inherent capabilities of an object[2] . The capabilities specified in an addon are lost, when the addon is deleted. Since an addon only defines an extension of an object class, instances of an addon do not exist.

Addons allow the capabilities of an object to evolve over time, like those of an object in the real world. During its life-cycle an object is created, acquires and loses relations and consequently gains and loses attributes, methods, rules and lifecycles. An example is an employee that has different capabilities in different jobs.

A class can also generalise other classes. There are no instances of a generalisation, since the class of an object defines its inherent, unchangeable properties. Generalisation captures commonalities between objects of different classes. It is needed to correctly specify certain relations. An example is the notion of a legal entity. Both companies and persons are legal entities, but no object exists that is only a legal entity. A lot of relations, however, are between legal entities. Hence, we need the ability to specify such generalisations in DEGAS.

## 3.2 Relations

Autonomous objects are entities with an independent existence. The interesting thing in a database, however, are the relations between the objects. Relations between objects are objectified in our data model, as is often the case in information systems modelling methods such as NIAM [26]. The motivation for this is a formal view of a relation as a kind of contract. Moreover, it ensures that relations can engage in relations themselves.

Before two objects enter a relationship, certain preconditions will have to be satisfied. For example, if two persons wish to marry, both must be of a different sex and unmarried. Likewise, the termination of a relationship is subject to restrictions. In a lot of relations we need to store data and behaviour of the relation. An obvious example is the bank account relation between a bank and its clients. This information and the capabilities to handle termination of

[2] An object with introspection might keep track of its capabilities to determine which capabilites are permanent and which are transient. A DEGAS object, however, is not equipped with introspection

the relationship are stored in a relation object. The capabilities to handle the initiation of a relationship, including the creation of the relation object, can be found in the corresponding relation class object.

An object that engages in a relation is extended using the addon mechanism, because it must gain capabilities to deal with the relationship. For example, once a person has a bank account, he can transfer money to other bank accounts or withdraw money through a cash dispenser.

### 3.3 Object Model

The three sorts of entities described above lead to a three-layered structure of a DEGAS database. At the lowest level we find the object instances. These are objects and relation objects with their addons. Addons do not have a separate existence, since they only define an extension of an autonomous object. Each class of objects is represented by a class object. These are again typed by the three metaclasses: objects, relation objects and addons. These three layers are depicted in Figure 2 and can be characterised as follows:

**Instance Level** At this level we have the representation of the real world of our information system. Here objects such as persons, banks and bank accounts can be found.

**Class Level** Each type is represented at this level by a class object for every object class, relation object class and addon class. Class objects handle object creation and keep track of the objects in their class. Class objects are DEGAS objects without the ability to engage in relations.

**Metaclass Level** The metaclasses are also represented by objects in the system. This is the highest level in the system. The presence of metaclass objects facilitates schema evolution by creating and destroying classes in analogy to the creation and destruction of objects by class objects.

## 4 DEGAS by example

In this section we introduce the DEGAS concepts by modelling an application. Challenging applications to model are those with high dynamics. An application with fast changing data and rapidly evolving relations is the stock market[3]. 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.

Let us briefly describe the example in more detail. Companies are owned by persons. A person can buy shares and sell them. He can subscribe to a newspaper specialised in news about companies of interest. 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

[3] Our example is a simplification of the stock market in the Netherlands.

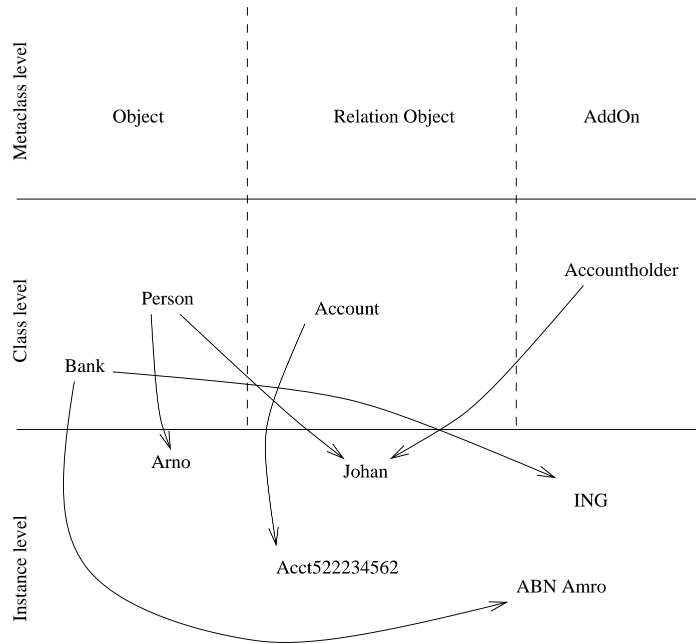


Figure 2: Structure of the Object Model

selling orders and to try to match these. This is specified by the following DEGAS definition of attributes and methods of an object class Marketmaker. 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.initiateMarketMaker
   }
   takeBuyOrder = {
      DemandClass.initiateMarketMaker
   }
   makeMarket = {
      ExtendSupplyDemand
   }
```

This defines the basic properties and actions, but we know more about the marketmaker. Therefore, an object includes a lifecycle description. The lifecycle of a marketmaker consists of taking buying and selling orders. If both actions have occurred in an arbitrary number and sequence, 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 register supply and demand, and clear the market if both are

present. The following rule, that completes the definition of the marketmaker object, specifies this behaviour:

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

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.initiatePerson(paper)
   }
   useNews = {
      ExtendInformedOwner
   }
Lifecycles
   (tryToBuy)*
   ((extendShareholder||extendInformedPerson);useNews)*
Rules
   On (extendShareholder||extendInformedPerson) 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. Thus, in DEGAS we are able to control access to an object's methods in greater detail than in, for example, C++ [37], where the only distinction is between private, public and friend 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. In this example, a person who becomes a shareholder gains capabilities to sell the shares again. This is specified in the shareholder addon:

```
  Addon Shareholder
  Extends Person
  Attributes
    share : oid
  Methods
    tryToSell(company:string, number:integer, minPrice:real) = {
      SupplyClass.initiateShareholder(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. A further description of the way relations are established can be found in Section 9.

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)(t1);goodNews(company)(t2)
   if t2 − t1 ≤ 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 : 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
```

The diagram in Figure 3 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.

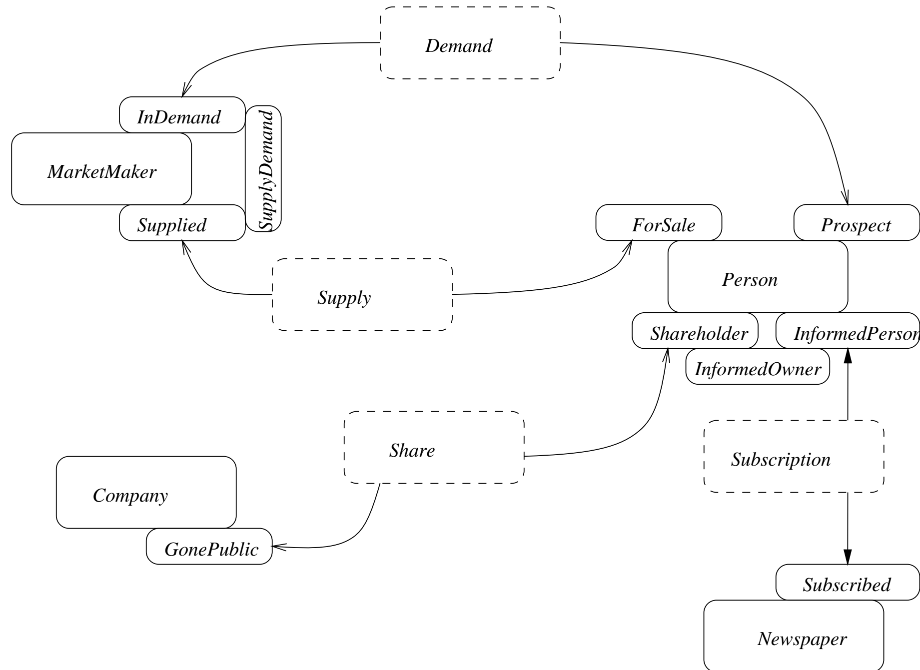


Figure 3: The DEGAS model for a financial market

# 5 Syntax of an Autonomous Object

In this section we give the syntax of the DEGAS data model. It is given as a BNF grammar. Symbols from the DEGAS language are printed in bold font, like the following: **Object**

**Object** The definition of a class has five parts, the header and four sections for the definition of the attributes, methods, rules and lifecycles.

$$\begin{aligned} \langle \text{Class} \rangle \quad \longrightarrow \quad & \langle \text{ClassHeader} \rangle \\ & \langle \text{AttributeSection} \rangle \\ & \langle \text{MethodSection} \rangle \\ & \langle \text{RuleSection} \rangle \\ & \langle \text{LifecycleSection} \rangle \\ & \langle \text{ClassEnd} \rangle \end{aligned} \tag{1}$$

Before we proceed with the class definition, we postulate the presence of the following disjoint sets from which terminals are taken:

| | |
|---|---|
| A set of basic types | ⟨*BasicType*⟩ |
| A set of basic functions | ⟨*BasicFunction*⟩ |
| A set of constant values for each basic type | ⟨*BasicValue*⟩ |
| A set of Boolean functions on the basic types | ⟨*Condition*⟩ |
| A set of attribute identifiers | ⟨*AttributeName*⟩ |
| A set of parameter identifiers | ⟨*ParameterId*⟩ |
| An ordered set of label identifiers | ⟨*LabelName*⟩ |
| A set of method identifiers | ⟨*MethodName*⟩ |
| A set of variable names | ⟨*VariableName*⟩ |
| A set of class names | ⟨*ClassName*⟩ |

The set of basic types includes at least the type *Oid* of object identifiers. The domains of the types are defined in the following section on semantics, because of the requirements subtyping poses on them. Basic functions are defined on the basic types or on a Cartesian product of basic types. The set of basic functions is closed under composition.

**Types** The basic types are used to construct complex types. Constant values can be used in expressions. The possible type constructs are power types and tuple types.

⟨Type⟩ ⟶ ⟨BasicType⟩ | $\mathcal{P}$ ⟨Type⟩ | ⟨TupleType⟩ | ⟨ClassName⟩

⟨TupleType⟩ ⟶ **(** ⟨FieldList⟩ **)** (2)

⟨FieldList⟩ ⟶ ⟨Field⟩ | ⟨Field⟩ **,** ⟨FieldList⟩ (3)

⟨Field⟩ ⟶ ⟨LabelName⟩ **:** ⟨Type⟩ (4)

**Class Header** The class header indicates the place of the class in the type structure. It defines the metaclass and a possible superclass. For relation object classes it defines the partners of the relation. In the definition of an addon class, it gives the class it extends.

⟨ClassHeader⟩ ⟶ **Object** ⟨ClassName⟩ (5)

⟨ClassHeader⟩ ⟶ **Object** ⟨ClassName⟩ **generalises** ⟨ClassList⟩ (6)

⟨ClassHeader⟩ ⟶ **Object** ⟨ClassName⟩ (7)
**Relation** ⟨ClassList⟩

⟨ClassHeader⟩ ⟶ **AddOn** ⟨ClassName⟩ (8)
**Extends** ⟨ClassName⟩

⟨ClassList⟩ ⟶ ⟨ClassName⟩ **,** ⟨ClassList⟩ | ⟨ClassName⟩ (9)

An example of an object class that generalises other object classes is:

**Object** LegalEntity **generalises** Person, Company

After the class header the capabilities of the class are defined. There are no syntactical differences between the metaclasses in the sections of a class definition.

**Attributes** Declaration of attributes is straightforward using the types defined above. Every (relation) object class has an attribute *this* : *Oid* containing the object identifier, which cannot be changed by the programmer. An addon does not have an identifier, because it is not an autonomous object.

⟨AttributeSection⟩ ⟶ **Attributes** (10)
⟨AttributeList⟩

⟨AttributeList⟩ ⟶ ⟨AttributeDecl⟩ | ⟨AttributeDecl⟩ **,** ⟨AttributeList⟩ (11)

⟨AttributeDecl⟩ ⟶ ⟨AttributeName⟩ **:** ⟨Type⟩ (12)

**Methods** The methods of an object are defined in the method section of the class declaration. A method may either modify the object state or call other methods. A method call can be either to an internal method or to a method of another object. Modification of the object state can take place through assignments to attributes. In addition method calls or assignments can be executed simultaneously on all elements of a set-valued attribute.

Methods included in every (relation) object class are those to add and delete addons from an object. This is explained in more detail in Section 9.

| | | | |
|---|---|---|---|
| ⟨MethodSection⟩ | ⟶ | **Methods** | (13) |
| | | ⟨MethodList⟩ | |
| ⟨MethodList⟩ | ⟶ | ⟨MethodDecl⟩ | (14) |
| | | \| ⟨MethodDecl⟩ **,** ⟨MethodList⟩ | |
| ⟨MethodDecl⟩ | ⟶ | ⟨MethodName⟩ **(** ⟨ParameterList⟩ **)=** **{** ⟨StatementList⟩ **}** | (15) |
| ⟨StatementList⟩ | ⟶ | ⟨Statement⟩ | (16) |
| | | \| ⟨Statement⟩ **;** ⟨StatementList⟩ | |
| ⟨Statement⟩ | ⟶ | ⟨AttributeName⟩ **:=** ⟨Expression⟩ | (17) |
| | | \| ⟨MethodCall⟩ | |
| | | \| ⟨AttributeName⟩ **:=** ⟨MethodCall⟩ | |
| | | \| ⟨SetIteration⟩ | |
| | | \| **Return** ⟨Expression⟩ | |
| ⟨Expression⟩ | ⟶ | ⟨AttributeName⟩ \| ⟨PathExpression⟩ | (18) |
| | | \| ⟨BasicFunction⟩ \| ⟨BasicValue⟩ | |
| ⟨PathExpression⟩ | ⟶ | ⟨PathExpression⟩ **.** ⟨AttributeName⟩ \| ⟨AttributeName⟩ | (19) |
| ⟨MethodCall⟩ | ⟶ | [⟨PathExpression⟩] **.** ⟨MethodName⟩ **(** ⟨ActParamList⟩ **)** | (20) |
| ⟨SetIteration⟩ | ⟶ | **forall** ⟨VariableName⟩ **in** ⟨AttributeName⟩ | (21) |
| | | **where** ⟨Condition⟩ | |
| | | **do** **{** ⟨StatementList⟩ **}** | |
| ⟨ActParamList⟩ | ⟶ | ⟨ActParam⟩ \| ⟨ActParam⟩ **,** ⟨ActParamList⟩ | (22) |
| ⟨ActParam⟩ | ⟶ | ⟨ParameterId⟩ **=** ⟨Expression⟩ | (23) |

**Rules** The rule section defines the rules on the object. These are Event-Condition-Action triples as is usual in active database systems. Event expressions are basic process algebraic expressions. Complex expressions are defined using sequential composition (;), alternative composition (+), repetition (*), parallel merge (||) and non-occurrence (¬) operators. An event expression cannot start with a negation, since this would mean that an event has never allowed in the history of an object.

| | | | |
|---|---|---|---|
| ⟨RuleSection⟩ | ⟶ | **Rules** | (24) |
| | | ⟨RuleList⟩ | |
| ⟨RuleList⟩ | ⟶ | ⟨Rule⟩ \| ⟨Rule⟩ **,** ⟨RuleList⟩ | (25) |
| ⟨Rule⟩ | ⟶ | **On** ⟨Event⟩ | (26) |
| | | **if** ⟨Condition⟩ | |
| | | **do** ⟨Action⟩ | |

$$
\begin{array}{rcll}
\langle\text{Event}\rangle & \longrightarrow & \langle\text{MethodName}\rangle\ [\langle\text{ParameterList}\rangle] & (27)\\
 & & \mid \mathbf{(}\,\langle\text{Event}\rangle + \langle\text{Event}\rangle\,\mathbf{)} & \\
 & & \mid \mathbf{(}\,\langle\text{Event}\rangle\,;\,\langle\text{Event}\rangle\,\mathbf{)} & \\
 & & \mid \mathbf{(}\,\langle\text{Event}\rangle\,;\neg\,\langle\text{Event}\rangle\,\mathbf{)} & \\
 & & \mid \langle\text{Event}\rangle^{*} & \\
 & & \mid \langle\text{Event}\rangle \parallel \langle\text{Event}\rangle & \\
\langle\text{Action}\rangle & \longrightarrow & \langle\text{MethodCall}\rangle & (28)
\end{array}
$$

**Lifecycles** The lifecycle of an object instance is defined in the lifecycle section of the class definition. Lifecycles are guarded basic process algebraic expressions. The actions used are method calls and the guards are conditions on the object state. Please note that, despite their similarity, lifecycles and event specifications use different expressions. Event expressions in rules can use the negation operator, whereas a lifecycle cannot. This difference originates in the nature of rules and lifecycles. A lifecycle is a positive description of what an object is allowed to do. An event expression in a rule is basically a query on the history of an object.

$$
\begin{array}{rcll}
\langle\text{LifecycleSection}\rangle & \longrightarrow & \textbf{Lifecycles} & (29)\\
 & & \langle\text{LifecycleList}\rangle & \\
\langle\text{LifecycleList}\rangle & \longrightarrow & \langle\text{Lifecycle}\rangle \mid \langle\text{Lifecycle}\rangle\,\mathbf{,}\,\langle\text{LifecycleList}\rangle & (30)\\
\langle\text{Lifecycle}\rangle & \longrightarrow & \langle\text{MethodName}\rangle\ [\langle\text{ParameterList}\rangle] & (31)\\
 & & \mid \mathbf{([}\,\langle\text{Condition}\rangle\,\mathbf{]}\,\langle\text{Lifecycle}\rangle\,\mathbf{)} & \\
 & & \mid \mathbf{(}\,\langle\text{Lifecycle}\rangle + \langle\text{Lifecycle}\rangle\,\mathbf{)} & \\
 & & \mid \mathbf{(}\,\langle\text{Lifecycle}\rangle\,;\,\langle\text{Lifecycle}\rangle\,\mathbf{)} & \\
 & & \mid \langle\text{Lifecycle}\rangle^{*} & \\
 & & \mid \mathbf{(}\,\langle\text{Lifecycle}\rangle \parallel \langle\text{Lifecycle}\rangle\,\mathbf{)} &
\end{array}
$$

This grammar defines syntactically correct classes. However, more is needed to get a meaningful hierarchy. To get such a hierarchy we need uniqueness constraints and referential constraints.

**Uniqueness Constraints** These constraints state that there must be unique names for classes, attributes, labels and methods.

**Referential Constraints** The references to other entities in declarations must be correct. Too be precise:

1. All methods must be well-typed. All assignments and method calls must be correctly typed, i.e. all values must be of the same type, or a subtype of that type, as the attribute or parameter they are assigned to.

2. All classes referred to in declarations must exist in the class hierarchy.

3. A method call must have the right number of actual parameters.

# 6 Basic Semantics

The basis of the semantics of an autonomous object are types. In this section we will first discuss typing and interpretation of methods and attributes. After that we will discuss the execution of methods in an autonomous object.

## 6.1 Typing

Typing of attributes and methods in the autonomous data model follows [9]. We first give the semantic counterpart of the construction of the types at a syntactic level in Section 5.

A hermit type **1** is introduced to cater for functions that always return the same value. A hermit type consists of a single element. A function to **1** discards all its input values. A function from **1** always returns the same value.

**Definition 1** *Given a set of basic types $B$, a hermit type* **1***, an ordered set of unique labels $L$, the set of types $T$ is defined as follows:*

1. $\mathbf{1} \in T$
2. $B \subseteq T$.
3. $(\sigma \to \tau) \in T$, *if* $\sigma, \tau \in T$.
4. $\langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle \in T$, *if* $m \in I\!N$ *and for* $1 \leq i \leq m$ $\tau_i \in T$, $a_i \in L$ *and* $a_i < a_{i+1}$.
5. $\mathcal{P}\tau \in T$, *if* $\tau \in T$.
6. $\sigma \times \tau \in T$, *if* $\sigma, \tau \in T$.

An object definition, as well as an addon definition, defines an underlying type, a tuple with the attributes of the object.

**Definition 2** *To each object definition and each addon definition $D$ we can apply an operator $Type(D)$ that yields the underlying type $\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$ defined by $D$. References to other classes are cast to the type $Oid$. The underlying type of an object definition contains at least the attribute $this : Oid$.*

As an example, the underlying type of a Person object, defined in Section 4, it it is not extended by any addon is:

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

A subtyping relationship is defined on the types following Cardelli [10].

**Definition 3** *The subtyping relation $\leq : T \times T$ is defined as follows:*

1. *Let* $\sigma = (\sigma_1 \to \sigma_2) \in T$ *and* $\tau = (\tau_1 \to \tau_2) \in T$. *If* $\tau_1 \leq \sigma_1$ *and* $\sigma_2 \leq \tau_2$, *then* $\sigma \leq \tau$.
2. *if* $\sigma, \tau \in T$ *and* $\sigma \leq \tau$, *then* $\mathcal{P}\sigma \leq \mathcal{P}\tau$.
3. *if* $\sigma_1 = \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \in T$ *and* $\sigma_2 = \langle m_1 : \nu_1, \ldots, m_k : \nu_k \rangle \in T$, *such that* $\forall i \in \{1, \ldots, k\}, \exists j \in \{1, \ldots, n\} : m_i = l_j \wedge \tau_j \leq \nu_i$, *then* $\sigma_1 \leq \sigma_2$.

The domains of the basic types are given in the following definition.

**Definition 4** *With each basic type $\beta$ is associated a domain $D(\beta)$. For example $D(real) = I\!R$.*

The hermit type **1** consists of a single element.

**Definition 5** $D(\mathbf{1}) = \{\emptyset\}$.

The domain of each class is a set of object identifiers.

**Definition 6** *Let $C$ be a class and $Oid$ a infinite set of distinct object identifiers. The domain of $C$ is a subset of $Oid$, $D(C) \subseteq Oid$, such that:*

1. *If $C_1$ **generalises** $C_2$, then $D(C_2) \subseteq D(C_1)$.*
2. *If $C_1 \neq C_2$ and not $C_1$ **generalises** $C_2$ or $C_2$ **generalises** $C_1$, then $D(C_1) \cap D(C_2) = \emptyset$.*

Before we define the domains of the types, we need to postulate the domains of the basic functions.

**Definition 7** *Given a set of basic functions $BF$. For each $\sigma \to \tau$, we postulate the domain $D(\sigma \to \tau)$ containing the basic functions from $\sigma$ to $\tau$.*

The domains of the types are defined following [9], such that $D(\sigma) \subseteq D(\tau)$ if $\sigma \leq \tau$. We first define the predomains of the types:

**Definition 8** *For each type $\tau \in T$ the predomain of $\tau$, $D_p(\tau)$, is defined as follows:*

1. *The predomain $D_p(\mathbf{1})$ is postulated in Definition 5.*
2. *The predomains $D_p(\beta)$ of a basic type $\beta$ are postulated in Definition 4.*
3. $D_p(\mathcal{P}\tau) = \mathcal{P}D_p(\tau)$
4. $D_p(\langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle) = \{\langle l_1 : a_1, \ldots, l_n : a_n\rangle | a_i \in D_p(\tau_i)\}$
5. $D_p(\sigma \times \tau) = \{(s,t) | s \in D_p(\sigma) \wedge t \in D_p(\tau)\}$

*For the functional types $\sigma \to \tau$ the predomains are defined as follows:*

1. *The predomain $D_p(\sigma \to \tau)$ of a basic function $f : \sigma \to \tau \in BF$ is postulated in Definition 7.*
2. $D_p(\sigma \to \tau) = \{f \cdot g | \exists \rho \in T : f \in D_p(\rho \to \tau) \wedge g \in D_p(\sigma \to \rho)\}$
3. $D_p(\mathcal{P}\sigma \to \mathcal{P}\tau) = \{f^* | f \in D_p(\sigma \to \tau)\}$, *where* $f^*(A) = \{f(a) | a \in A\}$
4. $D_p(\langle \alpha_1 : \sigma_1, \ldots, \alpha_n : \sigma_n\rangle \to \langle \beta_1 : \tau_1, \ldots, \beta_m : \tau_m\rangle) =$
   $\{(f_1, \ldots, f_m) | f_i \in D_p(\rho_1 \times \ldots \times \rho_l \to \tau_i), \rho_i \in \{\sigma_1, \ldots, \sigma_n\}\}$
5. $D_p(\sigma_1 \times \ldots \times \sigma_n \to \tau_1 \times \ldots \times \tau_m) =$
   $\{(f_1, \ldots, f_m) | f_i \in D_p(\rho_1 \times \ldots \times \rho_l \to \tau_i), \rho_i \in \{\sigma_1, \ldots, \sigma_n\}\}$

From these predomains we derive the domains as follows:

**Definition 9** *For each type $\tau \in T$ the domain $D(\tau)$ is constructed as follows from the predomains:*

1. $D(\mathbf{1}) = D_p(\mathbf{1})$
2. *For a basic type* $\beta$, $D(\beta) = D_p(\beta)$
3. $D(\mathcal{P}\tau) = \mathcal{P}(D(\tau))$
4. *If* $\tau = \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$, *then* $D(\tau) = \bigcup_{\sigma \leq \tau} D_p(\sigma)$.
5. $D(\sigma \times \tau) = \{(s,t) | s \in D(\sigma) \wedge t \in D(\tau)\}$
6. $D(\sigma \to \tau) = \bigcup_{\sigma' \leq \sigma, \tau' \leq \tau} D_p(\sigma' \to \tau')$

Methods are typed as well in the autonomous data model. This is done through function types, like in TM/FM [8]. In this approach a method is a function mapping an object state and instantiated input parameters to a new object state and instantiated output parameters.

In our model, however, the underlying type of an object a method is defined on is not fixed. This is caused by the evolution of an object through the addon mechanism. For example, consider the attributes of a person object before and after extension by an account holder addon. Before the extension the attributes are given by the tuple $\langle$*Name : string, Birthday : date, Purse : integer*$\rangle$, while afterwards the attributes are the tuple $\langle$*Name : string, Birthday : date, Purse : integer, Account : Oid*$\rangle$.

The evolving type of an object means that typing of methods is a bit more complicated than in the case of a fixed type. To define the type of a method, we first have to define how tuple types can be combined. The combination of tuple types is of importance for the definition of the extension of an object through addons. The tuple type addition operator is defined as follows.

**Definition 10** *Given two tuple types* $T = \langle t_1 : \tau_1, \ldots, t_n : \tau_n \rangle$ *and* $S = \langle s_1 : \sigma_1, \ldots, s_m : \sigma_m \rangle$ *with* $\{t_1, \ldots, t_n\} \cap \{s_1, \ldots, s_m\} = \emptyset$, *the composition of these tuple types is defined as follows:*

$$T \otimes S = \langle u_1 : \nu_1, \ldots, u_{n+m} : \nu_{n+m} \rangle$$

*where* $u_i : \nu_i \in \{t_1 : \tau_1, \ldots, t_n : \tau_n, s_1 : \sigma_1, \ldots, s_m : \sigma_m\}$ *and* $\forall i, 0 < i < n+m : u_i < u_{i+1}$.

Because the type of an object is not fixed, a method can only operate on those attributes whose presence is certain. These are the inherent attributes of an object and those attributes defined in the same addon as the method itself.

In the typing of methods the absence of a fixed underlying type of an object can be solved by introducing a type variable representing the type context of a method [41]. It is used, for example, in TM/FM to correctly type inherited methods. In the DEGAS data model the relevant part of the type context for a method $M$ in an object $O$ is given by the class, or inherent type, of $O$ plus, if $M$ is defined in an addon $A$, the type $A$. The rest of the type context is represented by a type variable. Since the rest of the current type of an object is not of importance, it may be of any type.

**Definition 11** *A method $M$ with input parameters $in_1 : \tau_1, \ldots, in_n : \tau_n$ and output parameters $out_1 : \sigma_1, \ldots, out_m : \sigma_m$ defined in an addon $A$ that extends a class $C$ is typed:*

$$\forall \rho \in T : \Delta \otimes \rho \times \tau_1 \times \cdots \times \tau_n \longrightarrow \Delta \otimes \rho \times \sigma_1 \times \cdots \times \sigma_m$$

*where $\Delta = Type(C) \otimes Type(A)$*

To illustrate this consider the example of a person object

**Object** Person

that is extended through an addon

**Addon** AccountHolder
**Extends** Person

In the addon AccountHolder the following method is defined:

```
GetCash(amount:integer) {
    Purse := Purse + Account.giveMeMoney(amount)
}
```

This method takes a Person object extended with an AccountHolder addon and an integer yielding again a Person object extended with an AccountHolder addon. The typing of the method GetCash is

$$\forall \rho \in T : \text{GetCash} : \Delta \otimes \rho \times \text{integer} \longrightarrow \Delta \otimes \rho$$

where the type context of this method is $\Delta = Type(Person) \otimes Type(AccountHolder)$.

The type of a class or an addon includes both attributes and methods. It will be useful for a number of further definitions in the report to have these separate.

**Definition 12** *Given an object class or addon $T$:*

1. *$Attr(T)$ yields the set of attributes defined in $T$.*
2. *$Meth(T)$ yields the set of methods defined in $T$.*
3. *$Cycl(T)$ yields the set of lifecycles defined in $T$.*

## 6.2 Interpretation and Method Execution

Now that we have discussed the typing of methods, we proceed with the semantics of the execution of a method. The semantics of method execution is defined in terms of variant interpretations and the semantics of the basic functions. An interpretation for an object maps the object to a value from the domain of the underlying type. Please note that the interpretation of an object as defined in this subsection, is not a complete object state. In DEGAS an object's state consists of its complete history. The full definition of an object's state will be given in Section 8. There, we will see that an interpretation gives a valuation for an object's attributes relative to a point in time.

**Definition 13** *An interpretation $I$ of an object $O$ is a function $I : Oid \longrightarrow D(\tau)$, where $\tau = \langle \alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n \rangle$ is the underlying type of $O$. $I$ assigns values $k_i$ to each $\alpha_i$ such that $k_i \in D(\sigma_i)$ where $\sigma_i \leq \tau_i$ for $1 \leq i \leq n$.*

A variant interpretation relates two interpretations to each other. A variant of an interpretation $I(O)$ is denoted by $I(O)\{\alpha = v\}$. $I(O)\{\alpha = v\}$ is the same as $I(O)$, except for the value assigned to $\alpha$, which is $v$.

The semantics of method calls are defined in terms of a function $M$. We consider as given the semantics of the basic functions on the basic types. We start with the semantics of assignment, which is defined in terms of variant interpretations.

**Definition 14** *Given a statement $S$ and an interpretation $I(O)$ for object $O$ , $M(S, I(O))$ returns the interpretation of $O$ after execution of $S$ on $O$. The effect of an assignment statement $A$, denoted by $M(A, I(O))$, is defined by:*

1. *Let $a_i : \tau_i$ be an attribute of $O$ and $v$ a correctly typed basic value, then $M(a_i := v, I(O)) = I(O)\{a_i = v\}$.*
2. *Let $a_i : \tau_i$ be an attribute of $O$ and $BF(p_1, \ldots, p_n)$ a correctly typed basic function call, then $M(a_i := BF(p_1, \ldots, p_n), I(O)) = I(O)\{a_i = BF(p_1, \ldots, p_n)\}$.*
3. *Let $a_i : \tau_i$ be an attribute of $O$ and $m(p_1, \ldots, p_n)$ a correctly typed method call, then $M(a_i := m(p_1, \ldots, p_n), I(O)) = I(O)\{a_i = R(m(p_1, \ldots, p_n))\}$, where $R$ is a function yielding the return value of a method call as defined in Definition 19.*

Set iteration, as defined in Production 21 of the syntax definition, is defined on the semantical level as follows:

**Definition 15** *Let $SI$ be a set iteration $\langle A, C_\tau, S \rangle$, where $A : \mathcal{P}\tau$ is a set-valued attribute of $O$, $C_\tau$ is a condition on a variable of type $\tau$ and $S$ is a statement. Given an interpretation $I(O)$ on an object $O$ the effect of $SI$, denoted by $M(SI, I(O))$, is defined as*

$$M(SI, I(O)) = M(S, I(\{C_\tau(a) | a \in A\}))$$

*$C_\tau(a)$ denotes that $C$ is true for $a : \tau$ in analogy with $C(O)$.*

A set iteration $\langle A, C_\tau, S \rangle$ corresponds to the DEGAS statement:

**Forall** $a$ **in** $A$
**where** $C_\tau$
**do** $S$.

Applying a statement to all elements of a set simultaneously amounts to taking the map of a function on a set.

**Definition 16** *Let $S$ be a statement, $A : \mathcal{P}\tau$ a set-valued attribute and $I(A) = \{v_1, \ldots, v_n\}$ its interpretation, then*

$$M(S, I(A)) = \{M(S, I(v_1)), \ldots, M(S, I(v_n))\}$$

Statements can be combined to form compound statements. The components of a compound statement are executed in sequence.

**Definition 17** *Given two statements* $S_1$ *and* $S_2$*, the semantics of the execution of a compound statement* $S_c = S_1; S_2$ *is defined as:*

$$M(S_c, I(O)) = M(S_2, M(S_1, I(O)))$$

The semantics of executing a method on an object is defined in terms of the statements forming the method. The statements used in defining the semantics of method execution are statements with the actual parameters in place of the formal parameters.

**Definition 18** *Given a method* $m = S_1; \ldots; S_k$*, the effect of a method call* $m$ *on object* $O$ *with interpretation* $I(O)$ *is given by:*

$$M(m, I(O)) = M(S_1; \ldots; S_k, I(O))$$

Methods can also return values. This means that we get a return value, as well as the effects of a method call defined previously.

**Definition 19** *Let* $m = S_1; \ldots; S_k; \textbf{\textit{Return}}\ e$ *be a method with* $S_1, \ldots, S_k$ *statements and* $e : \tau$ *an expression. The effect of executing* $m$ *on object* $O$ *with interpretation* $I(O)$ *is given by:*

$$M(m, I(O)) = M(S_1; \ldots; S_k, I(O))$$

*The result of the method call* $R(m) : \tau$ *that is returned to the caller, is defined as*

$$R(m) = M(m, I(O))(e)$$

# 7 Communication between Objects

In this section we describe the semantics of communication between objects in Degas. After stating some basic assumptions, we define the semantics of the communication mechanism of Degas, namely method calls.

## 7.1 Basic Assumptions

For the definition of Degas we must make a number of assumptions about its environment. These are mainly about the computing environment of the system. For more information on these subjects the reader is referred to standard textbooks on operating systems [38] and networks [39].

**Concurrency** Degas lives in a computing environment that supports multiple processes. Multitasking is pre-emptive, i.e. a process does not have to hand back control to the operating system explicitly. An object in Degas runs as a single process. In other words, there is no parallellism within an object.

**Communication** Communication between objects never disappears. Either a message sent reaches its destination, or an error will be reported, if something goes wrong. In other words, messages cannot disappear without a trace.

## 7.2 Method Calls

The reception side of a method call is explained in Section 10.2. Therefore this Section will focus on the sending side of a method call. Method calls occur in two places in a DEGAS object, methods and rules. A further distinction can be made between calls to methods of another object and calls to methods within an object. For the discussion in this section it should be noted that a method either executes some basic modification of attributes, or calls other methods.

**Method calls within an object** The simplest case are calls made to methods of the object itself. For this case we do not distinguish between calls made in methods and in rules. Since a DEGAS object runs as a single thread or process, the execution of a method calls is sequential. This means that the semantics of a method call within an object is a standard procedure call semantics.

**Method calls to other objects** DEGAS objects do not share memory. Hence, communication between objects must take place through message passing. A message to another object occurs in the form of calling a method in another object. The questions to answer are, whether method calls are blocking or non-blocking, and what happens in case of failures.

The choices with regard to the semantics of method calls must be made with the idea of object autonomy in mind. This means that an object must not be manipulated by another object, unless this manipulation is explicitly specified. For example, this means that an object should not be able to force another object into idleness by not answering a message.

The argument of object autonomy leads to non-blocking message passing, unless explictly specified otherwise. Hence, a method call is a request to another object to execute an action. In DEGAS, there are two possibilities to obtain a confirmation of the result of an action. First, a method call may occur on the right hand side of an assignment. If an error occurs, the method is rolled back. Second, another method may serve as an explicit confirmation of an action. In this case we can use the DEGAS rule mechanism to take corrective action, in case the confirmation does not occur. For example, we can specify the rule:

**Rules**
**On** Method($t_1$);$\neg$Confirm
**if** $t_{now} - t_1 \geq 2$ min
**do** Object.MethodInQuestion

**Semantics of calling a method** We now complete the semantics of method execution, defined in Section 6.2. The effect of an assignment of the result of a method call is defined in Definition 14. The nonblocking nature of method calls means that the state of an object is not changed by a call to a method in another object.

**Definition 20** *Given an object $O_1$ with an interpretation $I(O_1)$, then the effect on $O_1$ of executing a statement $S_{mc} = O_2.MC$ containing a method call to another object $O_2$ is defined as follows:*

$$M(S_{mc}, I(O_1)) = I(O_1)$$

*The effect of $S$ on $O_2$ is that the method call $MC$ is appended to the method queue of $O_2$.*

Until now, we have assumed that no problems are encountered during the execution of a method call. We assumed that the underlying system always delivers messages correctly. Hence, an object only has to deal with errors that occur in the receiving object. Furthermore, it only needs to do so, if it blocks to wait for an answer.

There are two kinds of errors, that may occur when calling a method. First, a method call may not satisfy the lifecycle of the receiving object. Second, the receiving object may encounter an error from a third object. In both cases, an error is reported. This error mechanism is built in DEGAS. It does not have to be specified by the programmer. The reaction of the sending object to an error will be to roll back the method it is itself executing and return an error message to the sender.

**Reception of Method Calls** When a method call arrives at an object $O$, it is placed in the method queue of $O$. The method queue of an object is a FIFO queue. The execution of incoming method calls is discussed in Section 10.2.

# 8 Time in an Autonomous Object

The aspects of the semantics discussed in the previous section were all independent of time. Since DEGAS aims at the integration of a historical database with an active database, this section proceeds with the temporal semantics of the DEGAS data model. The requirement on the temporal functionality of DEGAS is twofold. First, we want the historic values of attributes to be available. Second, the method calls executed must be available in order to check rule triggering and satisfaction of lifecycles.

We first discuss time and clocks in relation to autonomous object. Then, we give the initial definition of the history of an autonomous object. After that, we define the constraints over a history to be a valid object history.

## 8.1 Clocks and Autonomous Objects

A key ingredient for the discussion of temporal data is of course a time line. We must consider the nature of time in our model. We will also take a brief look at the source of time in a database of autonomous objects.

We timestamp once only, so there is no distinction between valid and transaction time in this model. Since the timestamp is produced by the system, DEGAS uses transaction time. In our model, time in an individual object is discrete and linear. The discreteness of time does not pose problems for applications, since we can make the granularity of time sufficiently small for most uses in an implementation.

Timestamps are always local. In other words, an autonomous object only works with its own historical view of the world. An event it has seen earlier, happened earlier in its time. Two general rules govern the way timestamps are handed out by an autonomous object. The first rule is that time increase monotonically. Second, two events never get the same timestamp.

**Definition 21** *The clock of an object $O$ is a function $T_O : \mathbb{N} \rightarrow \mathbb{N}$ that takes as input an event counter and yields as output its current time. $T_O$ is an invertible function.*

These requirements on timestamps are rather loose. It honours one of the two requirements formulated by Lamport in [22]. These requirements were formulated with help of a clock function $C_i$ for process $P_i$:

1. If $a$ and $b$ are events in process $P_i$ and $a$ comes before $b$, then $C_i(a) < C_i(b)$.
2. If $a$ is the sending of a message by process $P_i$ and $b$ is the receipt of the message by process $P_j$, then $C_i(a) < C_j(b)$.

In the distributed environment of a DEGAS database, it will be very difficult to guarantee Condition 2, which constrains the relation between the clocks of different autonomous objects. If different objects use different clocks, possibly running at different speeds, a lot of clock synchronisation would have to take place. The other option would be to use one global clock for all objects, which compromises the autonomy of objects even more. Hence, we do not follow requirement 2 in the DEGAS model.

In the DEGAS data model the only assumption regarding the presence of clocks is, that a local unique timestamp is available, whenever one is needed. This way we do not need any further assumptions on the presence of a clock in an autonomous object. The source of timestamps for an autonomous object is left to the implementation.

## 8.2 Pre-history

The history of a DEGAS object records past states of the object. This means that it records the historical valuations of the attributes of the object. An object records valuations for the complete set of attributes of an object with a timestamp indicating the start of the interval when this valuation held for the object. In addition the cause of the state transition is recorded. These are always caused by method calls, since the only way to modify an autonomous object is through its methods. Please note, that the history of a DEGAS object is not replayable. The reason for this is that a value may depend on a result returned by another object. Object autonomy implies that an object can not give guarantees on the state of another object.

Before we give the definition of the pre-history of an autonomous object, we need to define how method calls are recorded in the state history.

**Nested Method Calls** State transitions are caused by the execution of methods. A method can operate directly on the attributes of an object, but can also call other methods. When does the object reach a new state in the latter case? If method $M_1$ executes actions *Act* before calling $M_2$, the changes made by *Act* will be visible to $M_2$. This implies that we should insert the state reached after *Act* into the state history. We would have to label the state transition with a part of the method.

To avoid such problems we have put the constraint on method definition that a method may either contain basic modifications of the attributes or calls to other methods. Thus, we have no problems with intermediate states during the execution of a method.

The state transitions are effectuated by those methods that are leaves in the calling trees. For example, in the calling tree depicted in Figure 4, the methods $m2$, $m4$, $m5$ and $m6$ are the methods that actually change attributes. In order to find the rest of the calling tree back in

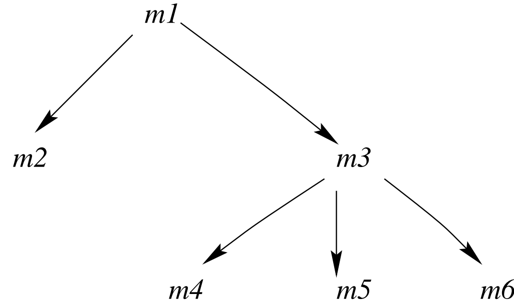


Figure 4: A method calling tree

the state history, we label the state transitions with nested method calls. For example, the call in Figure 4 will be represented in the state history by the following nested method calls:

$$m1(p_1, \ldots, p_m).m2(q_1, \ldots, q_n)$$
$$m1(p_1, \ldots, p_m).m3(r_1, \ldots, r_k).m4(s_1, \ldots, s_l)$$
$$m1(p_1, \ldots, p_m).m3(r_1, \ldots, r_k).m5(t_1, \ldots, t_j)$$
$$m1(p_1, \ldots, p_m).m3(r_1, \ldots, r_k).m6(u_1, \ldots, s_i)$$
$$m1(p_1, \ldots, p_m).m3(r_1, \ldots, r_k)$$
$$m1(p_1, \ldots, p_m)$$

**Definition 22** *Given a time $t$ and an object $O$. A nested method call $M$ is a sequence of method calls*

$$m_1(p_1, \ldots, p_k).m_2(q_1, \ldots, q_l). \cdots .m_n(r_1, \ldots, r_m)$$

*where*

1. *$\forall i, 1 \leq i < n$ a call to $m_{i+1}$ must occur in the body of $m_i$.*
2. *$\forall i, 1 \leq i < n$, if the subsequence $m_i(p_1, \ldots, p_k).m_{i+1}(q_1, \ldots, q_l)$ occurs in $M$, then $\forall q_j \in \{q_1, \ldots, q_l\}$, $q_j \in \{p_1, \ldots, p_k\} \cup Const$.*

*A nested method call is well formed if all method calls in it are well formed.*

The following function is used in a number of definitions in this paper. It yields the tail of a nested method call, which is the method actually being executed by that call. It is used to get an object's event history, which is used to check lifecycles and event expressions.

**Definition 23** *Given a nested method call $MC = m_1(p_1, \ldots, p_k). \cdots . m_n(q_1, \ldots, q_l)$. The function tail yields us the last method call in the sequence:*

$$tail(MC) = m_n(q_1, \ldots, q_l)$$

**State Pre-history** We now have all definitions we need to define a state of an object at a certain point in time. Following temporal database terminology this is called the snapshot state. A snapshot state records the time the object came in that snapshot state, a valuation for a tuple of attributes and the method call that brought the object into this snapshot state.

**Definition 24** *A snapshot state of an object $O$ is a quadruple $\langle t, \tau, I(\tau), MC \rangle$, where*

1. *$t$ is a timestamp giving the start time of the validity of this state.*
2. *$\tau$ is a tuple type $\langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle$, the underlying type of $O$ at time $t$.*
3. *$I(\tau)$ is the interpretation of $\tau$ in the interval starting at time $t$.*
4. *$MC$ a well-formed nested method call.*

The state history of an object records the snapshot states the object went through in the past. Not any sequence of snapshot states is a correct state history for an autonomous object. A state transition can only occur in the history, if it e.g. respects the constraints defined by the programmer of that object. Therefore we start with the definition of a state pre-history. A state pre-history is a sequence of snapshot states. In the following section we will define the constraints a state pre-history must satisfy to be a correct state history.

**Definition 25** *The state pre-history $SH$ of an object $O$ is a sequence of snapshot states*

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

$$SH(i) = \langle t_i, \tau_i, I(\tau_i), MC_i \rangle$$

*where*

$$\forall 0 \leq i \leq n-1 : t_i < t_{i+1}$$

An example of a state pre-history is the following:

$$\begin{array}{l} \langle 13:00:00, \langle balance : integer, week : integer, maxover : integer \rangle, \\ \qquad (balance = 1002, week = 130, maxover = 400), create(acct) \rangle \\ \langle 13:15:00, \langle balance : integer, week : integer, maxover : integer \rangle, \\ \qquad (balance = 902, week = 230, maxover = 400), withdraw(100) \rangle \end{array}$$

**Event Pre-history** In the definition of the DEGAS data model, we sometimes only need the historical events without state information. To deal with these definitions we define the event history of an object. The event history is defined in terms of the state history. The events that occur in an autonomous objects are the method calls that are executed by the object.

**Definition 26** *Given a state pre-history $SH = SH(0); SH(1); \ldots; SH(n)$, we define an event pre-history $EH = EH(0); EH(1); \ldots; E(n)$ of time-event pairs, where:*

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

### 8.3 Interpretation

In the DEGAS model we can refer to the state of an object at a certain point in time. To that end we define the notion of an interpretation of an object relative to the time.

**Definition 27** *Given a state history $SH = SH(0);\ldots;SH(n)$ for object $O$. The interpretation of $O$ at time $t$ $I(O)(t) = I(\tau_i)$, if*

$$\begin{aligned} SH(i) = \langle t_i, \tau_i, I(\tau_i), MC_i\rangle \\ \wedge \forall j > i, SH(j) = \langle t_j, \tau_j, I(\tau_j), MC_j\rangle : \\ t < t_j \end{aligned}$$

It is now straightforward to define the value of an attribute in an autonomous object at a certain point in time. For example, $balance(t)$ gives us the value of attribute $balance$ at time $t$.

**Definition 28** *The value of an attribute $a$ in object $O$ at time $t$ is denoted by $a(t)$. $a(t) = v_a$, if $I(O)(t) \models I(a:\sigma)(t) = v_a$.*

From the interpretations of all objects in a database we construct a model for a database relative to the time.

**Definition 29** *Let $db = \{o_1,\ldots,o_n\}$ be a set of objects. $\Gamma(db)(t) = \bigcup_{i=1}^{n} I(o_i)(t)$ is a pre-model for $db$.*

If all object references in a pre-model exist, it is a model for a database.

**Definition 30** *Given a set of classes $C$, a pre-model for a database at time $t$ $\Gamma(db)(t)$ is a model for $db$, iff*

$$\forall I(a:\sigma)(t), \sigma \in C, \exists o \in db : I(a:\sigma)(t) = o$$

From a model of a complete database we can get a model for the part of the database related to a specific object. To determine this part we need the notion of reachability. An object $O_1$ is reachable from another object $O_2$, if we can construct a path expression starting with an attribute in $O_2$ that yields a reference to $O_1$.

**Definition 31** *$\Gamma(O)(t) \subseteq \Gamma(db)(t)$ is the model induced by $O$. It is defined by*

$$\Gamma(O)(t) = \bigcup_{o \in R_O} I(O)(t)$$

*where $R_O$ is the set of objects reachable through path expressions from $O$ in $\Gamma(db)$.*

## 9 Relations and addons

This section gives a formal description of the way relations and addons are dealt with in DEGAS. First, we describe how objects engage in relations. After that, we explain how an addon can extend an object.

## 9.1 Engaging in Relations

In Section 3.2 we gave an informal description of the initiation of a relation. Here we will describe exactly how a relation between objects is set up. The only requirement DEGAS puts on the creation of a relation, is that the relation class object must create the relation object. The rest is left to the application programmer. This subsection illustrates some possible ways to set up a relation.

**Initiative** The initiative for a relation is taken by an autonomous object. It indicates its intention to enter a relationship by sending a message to the class object of the desired relation. It has the standard name *initiatePartner* in every relation class object. *Partner* is the class name of the partner sending this message, e.g. a person object will send an *initiatePerson* message to a relation class object. In the case of a relation between two objects of the same class, the initiate message will contain the name of the role in the relation. For example, to initiate a marriage relation a male person object will send a *initiateHusband* message. A *initiate* message can have parameters, for example containing the desired partner in the relation.

The first precondition of a relation is that all partners agree to engage in the relation. Hence, the first action of the relation class object on receipt of an *initiate* message is to check this. A relation can be established, if the relation class object has received an initiate message from all partners in the relation.

The reaction of the relation class object to an *initiate* message is dependent on the relation. In some cases one object takes the initiative for a relation and send an *initiate* message to the relation class object. Its reaction will be to send a message to the an appropriate partner for the relation. An example is found in the stockmarket example in Section 4. On receipt of an *initiate* message from a shareholder, the class object of the Supply relation will ask a marketmaker to take the order. This is specified by the following rule:

```
Rules
  On initiateShareholder
  do marketMaker.takeBuyOrder
```

In the specification of the marketmaker object we see that a marketmaker object returns an *initiateMarketMaker* message in the *takeBuyOrder* method. This means that it always accepts Supply relations. On receipt of the *initiateMarketMaker* message the Supply class object will go ahead to create the relation, if all preconditions are satisfied.

A variant of this example is, that the first candidate partner expresses the wish to engage in a relation with a specific object as a partner. The difference with the previous case is, that the relation class object will send a request to that specific object. An example of this is the following rule:

```
Rules
  On initiateFirst(desiredPartner)
  do desiredPartner.requestRelation(sender)
```

Another possibility is that the relation class just matches two objects that want to engage in a relation. This would be implemented by the following DEGAS rule:

**Rules**
**On** initiateFirstPartner||initiateSecondPartner
**do** instantiateRelation

A final possibility is that the partners in the relation have already agreed to engage in a relation. This case is not very different from the previous one. The only difference is that both parties indicate the desired partners.

**Rules**
**On** initiateFirstPartner(firstPartner,secondPartner)||
initiateSecondPartner(firstPartner,secondPartner)
**do** instantiateRelation

From this paragraph we learn that the initiative to engage in a relation may occur in a number of variants. All of these can be specified in the relation class object using the DEGAS rule mechanism.

**Preconditions** Besides the agreement of the partners, there may be other preconditions to a relation. These are checked by the relation class object. If the preconditions are satisfied, the relation can be created. The relation class object will send messages to the partners, instructing them to extend themselves with the appropriate addon. If this succeeds, the relation object will be instantiated.

To specify the preconditions of a relation, we can use the DEGAS lifecycle mechanism. A relation can only be created if the preconditions of the relation are satisfied. Hence, in the lifecycle of a relation class object these preconditions are the guard conditions to the action that creates the relation object. For an example, we again take the Supply class object:

**Lifecycles**
(initiateShareholder;initiateMarketMaker;
[*Preconditions*]instantiateRelation)*

This specification of the preconditions of a relation leaves us with the question, what happens if the preconditions are not satisfied. In this case, the relation class object must inform the prospective partners. In short, an instantiation action must be executed, if the preconditions are satisfied, while a cancellation action must be invoked, if the preconditions are not satisfied. Hence, the preconditions are better specified by rules.

**Lifecycles**
(initiateShareholder;initiateMarketMaker;
(instantiateRelation+cancelInitialisation))*
**Rules**
**On**initiateMarketMaker
**if** *Preconditions*
**do** instantiateRelation
**On**initiateMarketMaker
**if** *not Preconditions*
**do** cancelInitialisation

The final action in the creation of a relation is the instantiation of the relation object and to activate the addons of the partners. The extension of objects through addons is discussed in the next subsection.

**Termination** In a DEGAS database relations will be terminated at some point in time. Like the creation of a relation, the termination procedure is specified by the programmer. This specification, however, is located in the relation object instead of the relation class object. Usually, this will proceed analogous to the creation: One of the partners indicates that it wishes to end the relation. Dependent on the relation, the other partner will be asked whether he agrees. Possible conditions on the termination of a relation are again checked in the lifecycle of the relation object.

To conclude this discussion, we return to the Stock Market example from Section 4. Below is the complete DEGAS specification of the Supply relation class object in the stockmarket example. The attribute SupplySet contains all current instances of the Supply relation.

```
Object SupplyClass
Attributes
    SupplySet : 𝒫Oid
    candidateShareholder : oid
    candidateMarketMaker : oid
Methods
    initiateShareholder = {
        candidateShareholder = sender
    }
    initiateMarketMaker = {
        candidateMarketMaker = sender
    }
    instantiateRelation = {
        Create Relation Object
        Extend Shareholder
        Extend MarketMaker
    }
    cancelInitialisation = { candidateShareholder.cancelSupply
    }
Lifecycles
    (initiateShareholder;initiateMarketMaker;
        (instantiateRelation+cancelInitialisation))*
    On initiateShareholder
    do marketMaker.takeBuyOrder
    On initiateMarketMaker
    if Preconditions
    do instantiateRelation
    OninitiateMarketMaker
    if not Preconditions
    do cancelInitialisation
EndObject
```

The following script illustrates the messages exchanged in order to create a Supply relation is created between a shareholder $S$ and a marketmaker $M$ through a Supply class object $SC$.

| $S$ | $M$ | $SC$ |
|---|---|---|
| $SC$.initiateShareholder | | |
| | | $M$.takeBuyOrder |
| | $SC$.initiateMarketMaker | |
| | | *Preconditions satisfied* |
| | | instantiateRelation |
| | | $RelId$ = createSupply($S, M$) |
| | | $S$.extendForSale($RelId$) |
| | | $M$.extendSupplied($RelId$) |

## 9.2 Object Extension

An object can be extended by an addon. The effect of object extension on the type of an object Supply relation is described in Section 10.1. An object has an activation action for each possible addon. It always knows these actions, regardless of the presence of the addons.

**Definition 32** *Given an object $O$ of class $C$. $C$ has the set of addons $AddonSet(C)$. For each $Add \in AddonSet(C)$, $O$ has a method* ExtendName, *where* Name *is the name of Add.*

In the following discussion we will use the generic terms *ExtendAddOn* and *DropAddOn* to indicate addon activation and deactivation methods. Information on each addon is already present in an object. This means that the lifecycle with regard to the *ExtendAddOn* and *DropAddOn* is known by an object before it is extended by that addon. Thus, the lifecycle of an addon can be used to restrict *ExtendAddon* and *DropAddOn* actions. An *ExtendAddOn* action can be parameterised in order to initialise an attribute in the addon, for example containing the identity of the relation object involved.

The identity of the relation an addon is connected to is stored in a *key attribute*. Each addon *Add* has a key attribute $\kappa_{Add} : \mathcal{P}Oid$. If $\kappa_{Add} \neq \emptyset$, then addon *Add* is active. Thus, this leads to the following definition of the addon set of an object:

**Definition 33** *The set of active addons $Addons(O,t)$ in an object $O$ of class $C$ is defined by:*

$$Addons(O,t) \stackrel{def}{=} \{Add \in AddonSet(C) | \kappa_{Add}(t) \neq \emptyset\}$$

The activation of an addon means inserting an object identifier in the key attribute of that addon.

In the previous section we saw that an object is extended with an addon, when it enters a relationship. A relation between objects in an application can be $1-1$, $1-n$ or $m-n$. This raises the question what happens when an object engages in a relation more than once.

If an object can only engage in a relation $R$ once, the addon $A_R$ will specify in its lifecycle that *Extend* and *Drop* actions are only executed in closed pairs. Hence, there is always only one element in $\kappa_A$

**Lifecycles**
(ExtendAddon;DropAddon)*

The other possible situation is that an object can engage in a relation more than once. Hence, the lifecycle of the addon will specify that the *Extend* operator can be executed more than once:

**Lifecycles**
ExtendAddon*
$([\kappa \neq \emptyset]$DropAddon$)^*$

In this case the necessary addon may already be present, when an object engages in such a relationship. Hence, the only effect of an extension action is that an element is added to $\kappa_A$.

**Adding capabilities** An addon adds capabilities to an object. Existing ones cannot be modified. An addition of a named capability, i.e. attributes and methods, is not allowed if it causes a name clash.

**Attributes** can be added by addons.

**Methods** can be added by addons. Methods cannot be modified in an addon. The only way to add functionality to a method of an object is through the definition of a rule. Suppose we want to extend a method $\mu$, then we can add a rule that triggers on $\mu$.

**Rules** can be added by addons. Analogous to the extension of a method, we can only extend the action of a rule by providing an additional rule that triggers on the same event.

**Lifecycles** can be added to an object's set of lifecycles. Added lifecycles are treated the same as the lifecycles in the original object, described in the next section.

# 10 From pre-history to history

We now have everything we need to define the constraints a pre-history must satisfy to be a valid object history. In part these constraints are part of the the DEGAS data model. Other constraints on the history are the dynamic lifecycles defined by the programmer of the object.

## 10.1 Type Evolution

The type of an object can change over time through the addon mechanism. This means that the attributes we see in the state history can vary over time. The variation is limited by the addons defined for the inherent type of the object.

If the type of an object is extended this must be done by an *extend* action. This is stated in the following constraint:

**Constraint 1** *The type of an object can only be extended through the addition of an addon. Given a state pre-history $SH = SH(0); \ldots; SH(n)$ of an object $O$ of class $C$:*

$$\begin{array}{l} \forall i, 0 \leq i < n \\ \quad SH(i) = \langle t_i, \tau_i, I(\tau_i), MC_i \rangle \\ \quad SH(i+1) = \langle t_{i+1}, \tau_{i+1}, I(\tau_{i+1}), MC_{i+1} \rangle : \\ \quad\quad \tau_i > \tau_{i+1} \\ \quad\quad \Rightarrow \\ \quad\quad \exists \alpha \in AddOnSet(C) : \tau_i \otimes Type(A) = \tau_{i+1} \wedge MC_{i+1} = ExtendAddon_\alpha \end{array}$$

*where $ExtendAddon_\alpha$ is the activation action associated with $\alpha$.*

Likewise the loss of attributes must be through an action to drop an addon.

**Constraint 2** *The type of an object can only be limited by dropping an addon. Given a state pre-history $SH = SH(0); \ldots; SH(n)$ of an object $O$ of class $C$:*

$$
\begin{aligned}
&\forall i : 0 \leq i < n \\
&\quad SH(i) = \langle t_i, \tau_i, I(\tau_i), MC_i \rangle \\
&\quad SH(i+1) = \langle t_{i+1}, \tau_{i+1}, I(\tau_{i+1}), MC_{i+1} \rangle : \\
&\qquad \tau_i < \tau_{i+1} \\
&\qquad \Rightarrow \\
&\qquad \exists \alpha \in AddOnSet(C) : \tau_i = \tau_{i+1} \otimes Type(\alpha) \wedge MC_{i+1} = DropAddon_\alpha
\end{aligned}
$$

*where $DropAddon_\alpha$ is the deactivation action associated with $\alpha$.*

In addition we have the constraint that the type of an object must always correspond to its set of active addons.

**Constraint 3** *Given a state $S = \langle t, I(\tau), MC \rangle$ for an object $O$ at time $t$,*

$$
\begin{aligned}
&A = AddOns(t, O) \\
&\Rightarrow \\
&\tau = Type(Class(O)) \otimes \textstyle\bigotimes_{a \in A} Type(a)
\end{aligned}
$$

## 10.2 Method and Rule Execution

Execution of a method is the only way to bring an autonomous object from one state to another. Therefore, the state of the object must be the result of the application of a method of the object to the previous state. In addition, any method call must satisfy the lifecycle of the autonomous object.

For the semantics of method and rule execution we need to check whether an object satisfies a condition. A condition can contain both local attributes and attributes in other objects through path expressions.

**Definition 34** *A condition $C$ defined in an addon $A$ extending class $C$ is a boolean function*

$$\forall \rho \in T : Type(C) \otimes Type(A) \otimes \rho \longrightarrow \{true, false\}$$

In further discussion of Degas we will need the notion that a condition holds on an object at a certain time.

**Definition 35** *$C$ is satisfied by $O$ at time $t$, denoted by $C(t, O)$, iff $\Gamma(O)(t) \models C$, where $\models$ denotes the standard logical inference relation [12].*

Execution of methods and rules must conform to the lifecycles on the object. In addition rules are triggered by the contents of the event history. As we saw above, lifecycles are regular event expressions where an event can be guarded by a condition. Lifecycles in a Degas object are similar to the way the dynamics of an object is modelled using finite automata in OMT [29].

**Definition 36** *Given an object $O$ of class $C$, a lifecycle is a guarded basic process algebraic expression where the event alphabet $\mathcal{M}$ is the set of methods defined on $O$.*

$$\mathcal{M} = Meth(C) \cup \bigcup_{\alpha \in AddOnSet(C)} Meth(\alpha)$$

*and the guard conditions are conditions as defined in Definition 34.*

The semantics of lifecycles is formulated in process algebraic terms [7].

**Definition 37** *Suppose we have an object $O$ with the following lifecycle definition:*

***Lifecycles***
$C_1$
$C_2$
$\vdots$
$C_n$

*Then $O$ follows the process:*

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

*with communication function $\gamma$ defined by: $\forall \alpha \in \mathcal{M} : \gamma(\alpha, \alpha) = \alpha$, where $\mathcal{M}$ is the event alphabet of $C$ as defined in Definition 36.*

In process algebra a communication function $\gamma$ 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.

To define what methods are allowed to execute, we need to know the set of lifecycles of an object relative to time.

**Definition 38** *Given an object $O$ of class $C$. The set of lifecycles at time $t$ is defined as follows:*

$$Cycl_O(t) = Cycl(C) \cup \bigcup_{\alpha \in AddOns(t,O)} Cycl(\alpha)$$

The set of lifecycles of an object is translated to a single process expression. Against this process method execution will be checked.

**Definition 39** *Given the set of lifecycles of an object $O$ at time $t$ $Cycl_O(t)$. The lifecycle of $O$ at time $t$ is given by:*

$$LC_O(t) = |_{C \in Cycl_O(t)} C$$

*with communication function $\gamma$ defined by:* $\forall \alpha \in \mathcal{M} : \gamma(\alpha, \alpha) = \alpha$, *where* $\mathcal{M}$ *is the event alphabet of* $C$ *as defined in Definition 36.*

A finite automaton is sufficient to match the event history with an event expression or a lifecycle.

**Proposition 1** *An event expression can be implemented by a finite state machine with conditions on the transitions. Such an automaton will be referred to as a lifecycle-checking or event-checking automaton in following definitions.*

**Proof** Follows from the fact that the event expressions are a regular language [23]. The transitions in a lifecycle-checking automaton are labelled by the preconditions and the method names. A lifecycle checking automaton is brought to the next state by a method execution. In an event-checking automaton they are labelled by an event name only. An event-checking automaton parses the event history for an event expression. The non-occurrence of events can be checked by providing transitions to non-accepting states for those events that are not allowed. □

We give an example of the automaton for an event expression incorporating a non-occurrence operator $\neg$. The automaton to parse the event expression $A; (B; \neg C; D)^*$ is depicted in Figure 5. State 2 is the accepting state of the automaton.

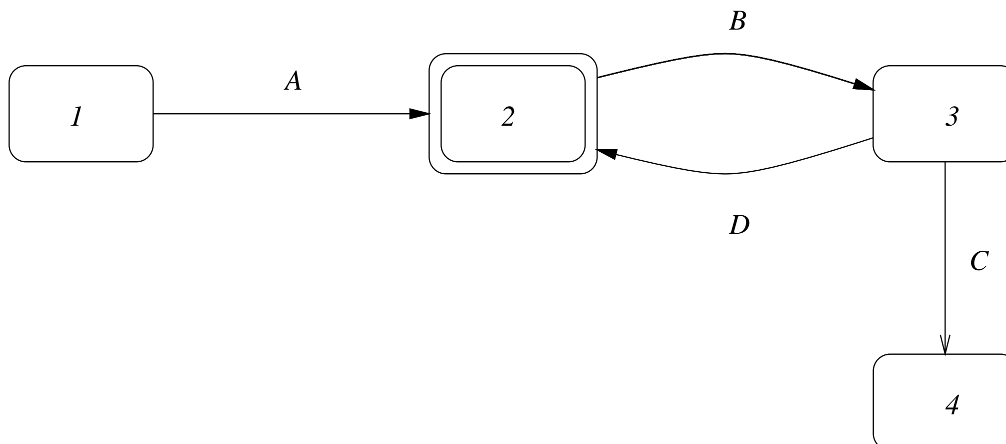


Figure 5: Event checking automaton

A method is executed if it does not violate the lifecycles imposed on the object. This means that the object state must satisfy the, possibly empty, precondition given by the lifecycle. In addition the method call in combination with the event history must match the event expression given. If this is true, the method call is executed and appended to the event history. If the method call does not satisfy the lifecycle of an object, it is removed from the method queue.

**Definition 40** *A nested method call* $MC = m(p_1, \ldots, p_m) \cdot \cdots \cdot m(q_1, \ldots, q_k)$ *is executed on an object* $O$ *with state history* $SH = SH(0); \ldots; SH(n)$ *at time* $t$*, iff the lifecycle checking automaton induced by* $LC_O(t)$ *is in a state with an outgoing transition labelled with a method*

*name $\mu$ and a condition $C$, such that $m = \mu$ and $C(t, O)$. The resulting new state of the object is*

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

This requirement on method execution can be translated to the following constraint on the state history of an object.

**Constraint 4** *Given a state history $SH = SH(0); \ldots; SH(n)$ of object $O$. Then each nested method call $MC$ occurring in $SH$ at time $t$ must follow the lifecycle $Cycl_O(t)$.*

As explained before, rules in DEGAS use the event-condition-action format.

**Definition 41** *A rule $R$ is a triple $\langle E, C, A \rangle$, where $E$ is an event expression as defined in Production 27, $C$ a condition as defined in Definition 34 and $A$ a method call as defined in Production 20.*

Recall that a rule $R$ which appears in the semantics of DEGAS as $\langle E, C, A \rangle$ is written in the DEGAS syntax as:

**On** $E$
**if** $C$
**do** $A$

Before we define the execution of rules, we first give an informal sketch of the execution model of an autonomous object. Basically, the object executes a cycle of two activities. There is a queue of method calls waiting to be executed. The object takes a method call from this queue, checks if it is allowed by the lifecycles, and if so executes it. If the method call does not agree with the object's lifecycle, it is not executed and removed from the queue. During the execution of a method a number of rules may be triggered. These are collected in a set of triggered rules. After a method has finished, a rule is picked at random from this set for execution. After the execution of this rule we again construct a set of triggered rules to pick a rule from. If no more rules are triggered, we start again with method execution.

From this informal description, we learn that 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 the method has finished, rules are picked at random from this set. If the condition of the picked rule holds, the rule is executed.

A rule is triggered by an event occurring in the event history as a result of a method call.

**Definition 42** *Given a state pre-history $SH = SH(0); \ldots; SH(n)$, an event pre-history $EH = EH(0); \ldots; EH(m)$ and a nested method call $MC = \mu_1(p_1, \ldots, p_k). \cdots .\mu_l(p'_1, \ldots, p'_{k'})$ and a time $t$. Rule $R = \langle E, C, A \rangle$ is triggered at time $t$, if $E$ parses the new event history $EH; \langle \mu_l(p'_1, \ldots, p'_{k'}), t \rangle$ correctly.*

After each execution of a method the set of triggered rules is constructed. This means that the execution of a complex method is interrupted by the execution of rules triggered by the methods it calls.

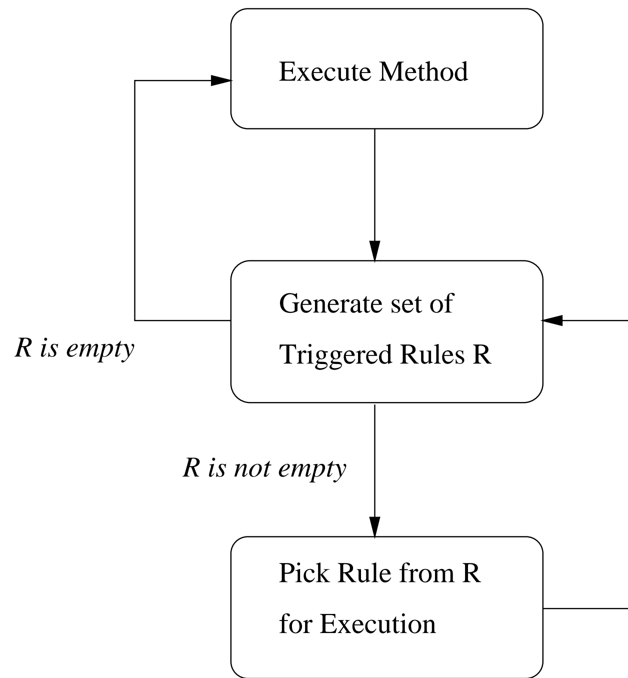


Figure 6: Execution Cycle of an Autonomous Object

**Definition 43** *If a rule $R$ is triggered by the execution of a method $\mu$ and the condition of $R$ holds, it is added to the set of rules triggered by $\mu$ $\mathcal{R}_\mu$.*

After a method has finished, one rule is picked for execution from the set of triggered rules at random. If its action satisfies the lifecycle of the object, it is executed. Otherwise, another rule is picked for execution. During rule execution rule triggering continues. In other words, after each method executed as a consequence of a rule, a new set of triggered rules is constructed.

**Definition 44** *Given a rule set $\mathcal{R}_\mu$ after the execution of method $\mu$ 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 $\mathcal{R}$ at time $t$.*
3. *If it satisfies the lifecycles of the autonomous object, action $A$ is executed. Otherwise, discard $R$ and goto step 2 again.*
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.*

This execution model gives us flexibility in the way rules interact with the lifecycles. If the action of a rule cannot be executed, because it violates the lifecycles of the object, we have two options. The action can be discarded or it can be retried at a later time.

The former will be used in situations where only a timely reaction is useful. These can be found in applications in financial markets. For example, our reaction to a falling price of shares

is that we buy a number of them. This is only profitable if we do it immediately, because otherwise the price may already have risen again. The other strategy is of use if the action of a rule must always be executed once a rule has been triggered. Rules that maintain integrity constraints will use such a strategy.

Both strategies can be used in DEGAS through the non-occurrence operator $\neg$. Suppose an object must react to the occurrence of the event $A; B$ with an action $\mu$. If we only want an immediate reaction, we will use the standard behaviour and specify the rule as:

**On** $A; B$
**do** $\mu$

On the other hand, if we want the action always to be executed after the event, we will specify the following rule:

**On** $A; B; \neg\mu$
**do** $\mu$

### 10.3 The History

We have now defined all constraints that a state pre-history and an event pre-history must satisfy in order to be a correct state history and event history for an autonomous object. This is stated in the following definition.

**Definition 45** *A state pre-history $H(O)$ of an object $O$ is a state history for $O$, iff it satisfies the constraints defined in this section.*

## 11 Design Decisions

In this section we examine a number of design decisions that lead to the current DEGAS model. For these decisions, we will outline the alternatives and motivate our choice for the model.

### 11.1 Methods, Rules and Lifecycles

In DEGAS the behaviour of an object is specified by methods, rules and lifecycles. Here, we explain why we have chosen for a behavioural specification consisting of these three elements. To this end, we consider the unification of methods and rules and the incorporation of lifecycles into the method specification.

**Methods vs Rules** In the DEGAS model there are two possible causes for state transitions, methods and rules. Can these two mechanisms be unified and, if this is possible, why is this not done in DEGAS?

The unification of methods and rules can be achieved by viewing methods as rules. In this case an object will have a set of events. Instead of placing a method call, an event can be raised in an object. The binding of a state transition to an event is specified by rules.

Modelling all state transitions as rules has a superficial appeal of simplicity. This is lost, however, when we look at the rule execution model that would be needed. In DEGAS the

execution of a method is guaranteed, if it conforms to the object's lifecycle. The DEGAS rule execution model was chosen for its simplicity and generality, and does not guarantee the execution of all triggered rules. Hence, the rule execution model accompanying this alternative model would be more complex than the current one.

**Lifecycles vs Guarded Method Calls** The potential behaviour of a DEGAS object is specified by methods and lifecycles. A method defines an action an object can execute, while the lifecycle specifies the relations between execution of different methods.

A different viewpoint on the specification of potential behaviour in a DEGAS object might be to specify it in one element. A method of doing this is to use guarded methods. A guarded method definition is a method definition extended with a guard, which specifies when a method is allowed to execute. For example, we could specify a method as:

**Methods**
MethodName(parameter : parameterType) =
**Guard**
*Guard condition*
**Body** {
*Method body*
}

In addition, this would offer us the possibility of overloading a method name, such that different actions would be executed in different object states. Chosing which of the alternative method definitions to execute is not immediately obvious. Satisfaction of the guard condition is not the only selection criterium. Specificity of a condition is needed as a next step, but in general this will not yield us a single method definition to execute.

A big drawback of this approach, however, is the expression of sequencing of methods. We cannot give expressions representing the possible execution sequences. Instead we will have to give explicit state information in the specification of an object. In our view, this is undesirable in a database programming language.

Overloading a method name through guarded method definitions gives us the possibility to modify a method in case of object specialisation. Again, however, explicit state information has to be given to achieve this.

On the other hand, guarded process algebraic expressions, as used in DEGAS, allow us to express sequencing in an easy way, while retaining the facility to express preconditions to the execution of a method. Furthermore, the effects of object specialisation through addons on an object's lifecycle as chosen in DEGAS, are clear and well understood.

## 11.2 Composing Lifecycles

The composition of the complete lifecycle for a DEGAS object from the set of specified lifecycles can be done in a number of ways. These alternatives can be identified using a number of questions on the nature of a lifecycle specification.

**Composition** The first question is about the composition of lifecycles. The main issue is how to deal with the occurrence of a method in multiple lifecycles. Suppose a method $\mu$ occurs in lifecycles $C_1$ and $C_2$. If a call to $\mu$ is made, it can be a step in only one lifecycle, or in both of the lifecycles containing the action. These two alternatives can be formalised in process algebraic terms. Suppose we have an object $O$ with a set of lifecycles $\{C_1, C_2, \ldots, C_n\}$. In the first case, this will lead to the following compound lifecycle:

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

If, in the other case, multiple lifecycles consume the same action, there will be, in process algebraic terms, communication between the lifecycles. Thus, these are composed using communication merge:

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

where the communication function $\gamma$ is defined by $\forall \mu \in A : \gamma(\mu, \mu) = \mu$ with $A$ the alphabet of the lifecycle definition.

**Lifecycle specification in addons** The second issue in lifecycle composition is the way we deal with lifecycles specified in addons. It is obvious that we wish to treat these lifecycles on the same footing as lifecycles specified in an object. There are, however, problems related to the specification of lifecycles in addons. These originate in the question whether an addon is allowed to modify lifecycles of the original object. If this question is answered positive, how can an addon modify the lifecycles. Another problem in that case is what happens if two addons modify the same lifecycle.

The main requirement on lifecycle specification in addons is that addons must conform to the original object. In other words, the lifecycles specified in an addon must not violate the lifecycle of the original object. If $C_O$ is the lifecycle of the original object $O$ and $C_A$ the lifecycle of $O$ extended with addon $A$, we can define this using the process algebra abstraction operator as:

$$\partial_H(C_A) = C_O$$

where $H$ is the set of methods defined in the addon. This constraint must be satisfied by redefinition of lifecycles. In practical terms, this means that an addon can intersperse an original lifecycle with its own methods.

If we use the communication merge as a composition operator, we get redefinition for free. This is shown by the following example. 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:

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

To illustrate the potential conflicts of lifecycle redefinition, consider the following situation, where two addons try to modify an object's lifecycle. An example are constraints added to objects in a graphical database, as shown in [2]. 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:

$$A; C; B$$

Real problems only start if we add an second addon $A_2$ that desires an action $D$ to be inserted in the lifecycle of $O$:

$$A; D; B$$

The question is what the desired lifecycle of $O$ is, if it has both $A_1$ and $A_2$, and how this should be specified.

If we use parallel composition for lifecycles in addons, we get the following lifecycle for $O$:

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

The drawback of this behaviour is shown, if we take the viewpoint of addon $A_1$. It does not know of the existence of $D$. Hence, seen through $A_1$ $O$ might execute $A; B$ without $C$ occurring in between.

If we use communication merge, the result would be that $O$ follows:

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

This conforms to the original lifecycle of the $O$, but each occurrence of $A$ and $B$ also satisfies the lifecycles of the addons.

**Our choice** Adding lifecycles in addons gives the most intuitive results, if we chose communication merge as the composition operator for lifecycles. It gives us inherent redefinition when it is needed. If we would use the parallel merge, redefinition would not be possible, thus putting undesirable constraints on the design of addons. Note however, that we can still express the other way of merging lifecycles by explicitly using a parallel merge.

The behaviour with regard to the specification of lifecycles in addons is the main reason for our choice of communication merge with identity as communication function as the lifecycle composition operator in Degas.

### 11.3 The Addon Mechanism

In Degas we introduced the addon mechanism to deal with object specialisation. We identify two alternatives for the addon mechanism, namely multiple inheritance and adding capabilities one at a time. In addition, we discuss the relation between addons and roles to show that the Degas addon mechanism is a good platform to implement roles.

**Addons vs Multiple Inheritance** The extension of an object through an addon yields us a specialisation of the original object. The addon mechanism offers a number of advantages over using inheritance to specialise objects. An addon offers the possibility to extend the capabilities of an object without knowledge of other specialisations of an object. 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 [24].

The advantage of the DEGAS addon mechanism over multiple inheritance lies in its clarity and simplicity. It protects the designer from an explosion of the number of classes, while retaining the ability to specialise objects in a number of ways at the time.

**Adding capabilities one at a time** Another alternative to the addon mechanism is a facility to add capabilities one at a time. In this case an object will only offer primitives to add and delete attributes, methods, lifecycles and rules without the possibility to add a group of capabilities.

A facility to add capabilities one at a time clearly offers a lot of flexibility. It should be noted however, that the DEGAS addon mechanism does not exclude this. The only restriction imposed is the requirement that a method must occur in a lifecycle in order to be executed. In our view, this requirement is not a liability, but instead forces the designer to give a complete specification of the actions of a DEGAS object.

The DEGAS addon mechanism offers the possibility to group the capabilities added to an object. The advantage of a grouping mechanism is found in the fact that capabilities are very often added in groups. A prime example is the concept of a role in information systems design, as discussed below. This concept in information systems design must have a counterpart in the implementation platform of the information system.

The choice for the DEGAS mechanism, as it is introduced in this report, is motivated by its clarity compared to multiple inheritance, while retaining a grouping facility for adding and deleting capabilities. In addition, the presence of class objects for addons, as described in Section 3, gives us the possibility to query for objects that are extended with a certain addon.

**Addons and roles** One of the key advantages of the DEGAS addon mechanism is found in the observation that object specialisation is tied to the role of an object. An object is specialised in order to play a role. 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 [27] 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 [3]. 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 [44]. 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 between the analysis in [44] and DEGAS is, that DEGAS only distinguishes between inherent and transient capabilities of an object.

## 12 Related Work

**Temporal Databases** If we look at the temporal functionality offered by DEGAS, we see that the OODAPLEX model offers the same functionality in the temporal sense. In addition it offers derived values, because of its deductive foundation. The state history of an autonomous object is very similar to Ginsburg's object histories. The method calling sequence and its parameters are equivalent to the action and the external attributes in an object history. The computation scheme of an object also defines a set of constraints. The constraints defined in our model are a subset of those in Ginsburg's model. Ginsburg's constraints can be any boolean function on an object history. Constraints on autonomous objects are in Ginsburg's category of localisable constraints, since they only refer to a finite part of the history of an object.

**Active Databases** A number of active database systems are object-based, notably HiPAC [13], SAMOS [19] and Chimera [11]. SAMOS and Chimera offer encapsulation of rules with object, although this is not the exclusive location of rule definition. HiPAC treats rules as separate objects. Time is an important consideration in HiPAC, but in an entirely different way than in the DEGAS model. It is designed with real time applications in mind. Therefore deadlines for triggers and scheduling of their action is an area of much attention there.

Another approach concerned with time in active databases is the work on temporal triggers by Sistla and Wolfson [33]. These are condition-action rules, where the condition is formulated in two variants of a temporal logic, called the Future Temporal Logic (FTL) and the Past Temporal Logic (PTL). These logics specify relations between timestamped database states. The basic modal operators are `Until` and `Nexttime`, thus allowing for the specification of durations. Temporal triggers address the temporal part of condition specification, but do not involve event specifications. We can only specify an action to occur when two events happen

within a certain time interval by stating in the condition the effects of those events on variables in the database.

An older system that aims to integrate active rules into objects is MOKUM [28]. Like DEGAS it incorporates active elements in objects. Activity in MOKUM occurs in two elements. First, attribute definition allows derived data and constraint enforcement. Second, triggers occur in scripts, that define the lifecycle of an object. A MOKUM script is a representation of a finite state machine. State transitions in a MOKUM script are defined by triggers, that specify an action and a state transition to be executed on an incoming event. Since MOKUM is Prolog-based, there is no distinction between conditions and actions in a trigger. Another similarity between DEGAS and MOKUM is the facility to add types to objects. There is, however, no interaction between the scripts of different types of an objects, as is possible with DEGAS lifecycles.

**Specification Formalisms** Formalisms to reason relative to time can be found in the area of temporal logics. Arapis [4] has applied a logic named Propositional Temporal Logic to the specification of object behaviour. This logic is a modal logic where time is the relation between the worlds. The only temporal notions are "before" and "after". Modal operators have a semantics such as "sometime in the future" or "always in the future". This means that we cannot specify things like "within an hour from now", which is often needed in a rule or constraint definition. An advantage of temporal logic is that combining descriptions is very straightforward.

DEGAS uses process algebra [7] to specify the lifecycle of an object. We do not, however, include time in the lifecycle specification, although some types of process algebra also include a temporal dimension. Variants with discrete time [6] and with real time [5] have been formulated. These process algebras offer about the same expressiveness as the event expressions in our model. Every action is timestamped. However the notation used is less suitable for use in a database programming language. The main problem is the need for quantifiers to decently specify the dynamic constraints as used in the DEGAS model.

**Intelligent Agents** In artificial intelligence research there is a lot of interest in intelligent agents. An intelligent agent has a lot in common with an active object, although it tends to be more complicated. Two approaches are discernible in the research on agents. On one side a lot of effort is into formalisms to describe the reasoning done by agents [32]. The specific tasks of agents are not very important in this approach. In the other approach [1] we see the development of agents specific for one application. Usually these agents are meant to take one specific task, such as scheduling a meeting, out of the hands of the user. Such agents interact with the user and other users' agents. In both approaches, however, management of data is not a subject that receives a great deal of attention in this area. All combinations of agents managing their own data and agents sharing databases occur.

DEGAS objects are a very general kind of reactive agents. Rules and lifecycles are good concepts to model obligations and rights in relations between objects, as often studied in more formal approaches to agents. Together with the addon mechanism, the addition of rules and lifecycles in DEGAS to traditional object-oriented notions facilitates the programming of agents.

# 13 Conclusions and Further Research

In this report we proposed DEGAS, a temporal active data model based on autonomous objects. DEGAS offers a formal framework for highly distributed, component-based information systems. This is achieved by founding DEGAS on autonomous objects, which encapsulate their complete behaviour. Thus, we offer a complete integration of dynamic and static parts of an application.

DEGAS' relation and addon mechanism makes it especially useful for applications with a highly dynamic content, such as the stockmarket example given in this report. At any time, an object has only the capabilities it needs. This means that addons offer a clean mechanism to implement roles. Lifecycles for objects and addons and their interaction are well defined using process algebra.

The contribution of DEGAS to the field of active databases is its temporal element. The notion that the state of an object is its complete history gives DEGAS temporal database functionality. Furthermore, an object history is a good foundation for the rule execution model.

The next phase in the development of DEGAS will be the definition of a query language. The conceptual view is that a query in DEGAS describes a set of autonomous objects. The existence of such a query need not be limited to the time needed to find an answer. A prototype implementation of DEGAS is underway.

# References

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

[2] 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*, pages 1-16, Maastricht, The Netherlands, September 1995.

[3] 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.

[4] Constantin Arapis. Temporal specifications of object behavior. In B. Thalheim, J. Demetrovics, and H.D. Gerhardt, editors, *Proc. of the 3rd Symposium on the Mathematical Fundamentals of Database and Knowledge Base Systems (MFDBS91)*, pages 308-324, Berlin, Germany, 1991. Springer. LNCS 495.

[5] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. Technical Report CS-R9053, CWI, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1990.

[6] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. Technical Report 92/06, TU Eindhoven, Eindhoven, The Netherlands, April 1992.

[7] 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.

[8] Herman Balsters, Rolf A. de By, and Roberto Zicari. Typed sets as a basis for object-oriented database schemas. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, 1993.

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

[10] 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.

[11] 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.

[12] D. van Dalen. *Logic and Structure*. Springer, Berlin, Germany, 2nd edition, 1985.

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

[14] Umeshwar Dayal, Alejandro P. Buchmann, and Dennis R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database system. In *Advances in Object-Oriented Database Systems*, pages 129–143, Berlin, Germany, September 1988. 2nd International Workshop on Object-Oriented Database Systems, Springer.

[15] Oscar Díaz, Norman Paton, and Peter Gray. Rule management in object-oriented databases: A uniform approach. In Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 317–326, September 1991.

[16] 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.

[17] 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.

[18] Hector Garcia-Molina and Boris Kogan. Node autonomy in distributed systems. In Sushil Jajodia, Won Kim, and Abraham Silberschatz, editors, *Proc. of the Intl. Symposium on Databases in Parallel and Distributed Systems*, pages 158-166, Austin, TX, USA, 1988.

[19] 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.

[20] Seymour Ginsburg. Object and spreadsheet histories. Chapter 12 in [40].

[21] Eric N. Hanson. An initial report on the design of Ariel: A dbms with an integrated production rule system. *SIGMOD Record*, 18(3):12-19, September 1989.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July 1978.

[23] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, Englewood Cliffs, NJ, USA, 1981.

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

[25] 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.

[26] 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.

[27] 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.

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

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

[30] Edward Sciore. Using annotations to support multiple kinds of versioning in an object-oriented database system. *ACM Transactions on Database Systems*, 16(3):417-438, 1991.

[31] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183-236, 1990.

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

[33] A. Prasad Sistla and Ouri Wolfson. Temporal triggers in active databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):471-486, 1995.

[34] Richard T. Snodgrass (chair). Tsql2 language specification, 1994. Available by FTP from ftp.cs.arizona.edu.

[35] H.G. Sol and K.M. van Hee, editors. *Dynamic Modelling of Information Systems*, Amsterdam, the Netherlands, 1990. North-Holland.

[36] Michael Stonebraker. Triggers and inference in database systems. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, chapter 22, pages 297-314. Springer, 1986.

[37] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, USA, second edition, 1991.

[38] Andrew S. Tanenbaum. *Operating systems: design and implementation.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1987.

[39] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, NJ, USA, second edition, 1989.

[40] 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.

[41] C.C. de Vreeze. Formalization of inheritance of methods in an object-oriented data model. Technical Report M90-76, Dept of Computer Science, Universiteit Twente, The Netherlands, 1991.

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

[43] R.J. Wieringa. *Algebraic Foundations for Conceptual Models.* PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1990.

[44] 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.

[45] Gene T.J. Wuu and Umeshwar Dayal. A uniform model for temporal object-oriented databases. In F. Golshani, editor, *Proceedings of the 8th International Conference on Data Engineering*, pages 584-593. IEEE, 1992.