Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

A data model for autonomous object

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

Computer Science/Department of Algorithmics and Architecture

CS-R9539 1995

Report CS-R9527
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

A Data Model for Autonomous Objects

Johan van den Akker
Arno Siebes

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

Abstract

Developments in distribution and networking of computing power raise questions about the feasibility of centralised control in an information system. At the same time the move towards information systems ranging over a number of different organisations brings us systems where central control is not desired. This asks for systems built of components that can function independently under local control only. A move towards autonomous components can also clearly be seen in the area of active databases. Rules are seen as part of the behaviour of objects or relations between objects. Following the encapsulation principle, this leads to encapsulation of all rules with objects. The definition of such an object is independent of other objects.

To support these developments we defined the data model for autonomous objects proposed in this report. An autonomous object is an object with its own thread of control. The behaviour of an autonomous object is defined by methods, rules and dynamic constraints. The latter two refer to the complete history kept with the object. The semantics of a relation between objects is captured in relation objects. This enables us to represent arbitrary complex conditions on initiating and terminating relations and to have arbitrary actions taken on certain events. Dependent on the relations an object has, its capabilities will evolve. This is achieved through addons. An addon defines capabilities an object can be, temporarily, extended with.

Structure is brought into the mass of objects at the instance level through the objects at the class level. Class objects occur for all object classes, relation object classes and addons. Their function as an object container enables the approach of groups of objects, for example for queries.

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

Keywords & Phrases: database programming language, objects, databases, data model, autonomy, activity, object evolution, conceptual models.

Note: Johan van den Akker is supported by SION, the Foundation for Computer Science Research in the Netherlands through Project no. 612-323-424.

INTRODUCTION

The combination of production rules and object databases, developments in distributed computing and networking, and developments in the way information systems are used by organisations indicate a need for systems composed of autonomous components. A data model to support such systems is defined in this paper.

Roadmap First we will sketch the developments that motivate systems of autonomous objects. Then we will describe the fundamental concepts of the autonomous data model. After that we will give syntax and semantics of the model. Finally we will relate the concept of autonomous objects to existing research in other areas.

1. AUTONOMY IN A DATABASE

1.1 What is Autonomy?

The fundamental concept in the data model described in this paper is object autonomy. Autonomy is two-sided. First, all static and dynamic aspects of an object are encapsulated with the object itself. Secondly, the object is only subject to its own, local, control. Hence, there is no central control.This means that the notion of object autonomy is stronger than the notion of a object activity. A database of autonomous objects is a database where distribution is taken to an extreme. Conceptually every object has its own thread of control.

1.2 From Activity to Autonomy

An active database [23] 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, 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 [13, 14]. This approach places the 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 triggered in specific situation.

Examples of object-oriented DBMSs that offer encapsulation of rules in objects are SAMOS [15] and Chimera [10]. 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.

We take encapsulation to its extremes. *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* in such a system.

1.3 Motivation for Autonomy

1.3.1 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 are the emergence of massively parallel computer systems and the coupling of existing computer systems over networks. These have in common that any form of central control will pose a large amount of overhead on the system.

Massively Parallel Computers To further raise the performance of computer systems massive parallelism is seen as the most promising road to travel. A very important condition for the acceptance of such platforms for general use is the presence of DBMSs. A key problem in such a DBMS will be how to distribute data and execution over the available processors. Centralising such decisions will pose a lot of overhead on the system. Enough overhead to make it a considerable factor in the performance of such a system. Therefore we must consider distributing decisions in the system. If we do this for all centrally controlled aspects of the DBMS, we have made the components of the system autonomous.

Information Systems in Mobile Networks The development of powerful mobile computers and the spread of wireless communications will make 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. 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 interoperate, it will be very difficult to devise a scheme clever enough to keep up with the size of such a network and its continuously connecting and disconnecting nodes.

It seems a better idea to have inherent flexibility built into such a system. What we want in this situation is that an arbitrary collection of objects is a functioning database. The objects actually in the collection of available objects may vary over time, but there is no system "authority" that needs to keep up with these changes. In such a systems an object is autonomous, because it functions independent of other elements of the system.

1.3.2 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 increasing 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 will want to determine what information can be seen by outsiders. Although the components are under control of separate owners, we do want to be able to approach such a chain information system as a single 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.

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

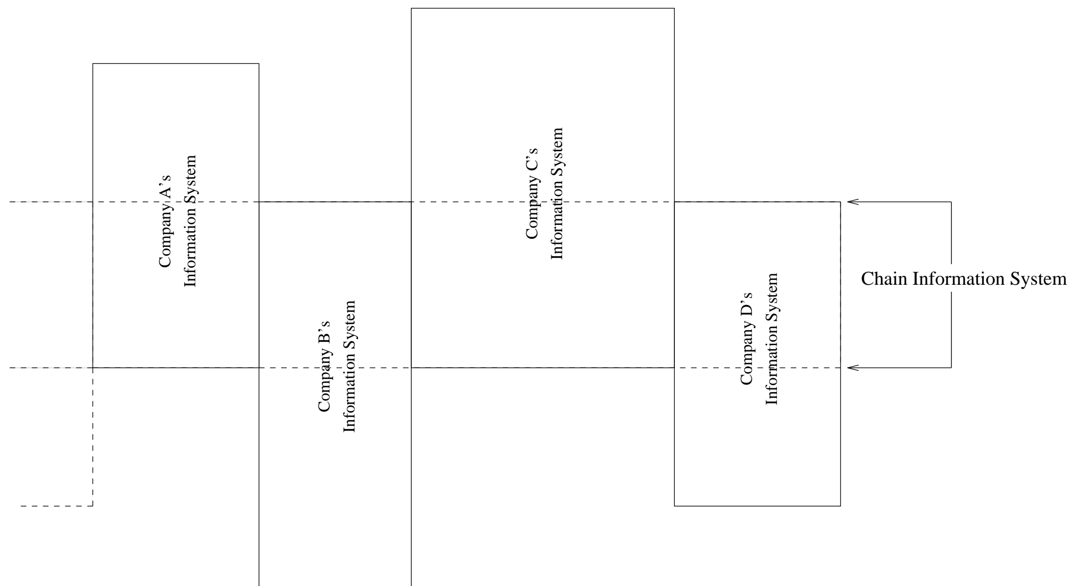

Figure 1: The integration of ISs in an inter-organisation IS

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 tume, but different sets of objects making up different databases simultaneously. The choice is agian between on one hand devising a clever scheme for a chain information system that gets data from different composing databases and on the other hand building in inherent flexibility at the object level. If an arbitrary set of objects were to function as a database, making objects visible to the other partners in a chain would be all it takes to integrate data 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.

Anarchic Information Systems An obvious example of a system with a lot of autonomy for the components is the World Wide Web (WWW). It could be described as an anarchic information system. Everyone can make information available as long as they conform to the conventions of HTML. If you want to change the information you offer or wish to move it to another place, you have complete freedom to do so without informing others. This may, for example, result in dangling references on other WWW pages. A scheme that enables us to approach the WWW as a single source of information, should leave the owners of pages complete autonomy in manipulating their pages.

Business Modelling The introduction of informations 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 [22, 26] 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.

1.4 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 1.2 we showed that object autonomy is the ultimate consequence of incorporating active rules in an object oriented database. Section 1.3.1 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 1.3.2 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.

We choose the object as the level of autonomy. We have made this choice because of its obvious advantages in modelling an information system. A model with autonomous objects also has the advantage of generality. In our model the complexity of the objects can be arbitrary. This means that the model can also be used for autonomous components at a higher level of abstraction. 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.

2. The Concepts underlying the Data Model

In the previous section we explained what object autonomy is and what developments promote object autonomy. In this section we describe how autonomous objects can form the foundation of a database system.

2.1 A Database of Autonomous Objects

In a database of autonomous objects everything known beforehand about an object is defined on the object itself. The behaviour of an object is defined by three components, methods, (dynamic) constraints and rules. 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. Dynamic constraints define what an object is *allowed* to do. Together methods and constraints 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.

In comparison with more traditional object-oriented databases, we add the rule component to an object's definition. Traditionally only potential behaviour is specified, whereas autonomous objects also contain their actual behaviour as far as that can be pre-determined.

Orthogonal to the active behaviour of objects is the evolution of their relations and capabilities. During its life-cycle an autonomous object develops just like anything in the real world from people to forms. An object is created, acquires and loses relations and consequently gains and loses attributes, methods, constraints and rules. For example, an employee is given different jobs over time where he can take different actions. Another example is an order that is processed by an organisation. The order will be represented by an autonomous object in the information system. Different parts of the company will successively manipulate the order. The physical processing of the order will be reflected in the relations the order object has. When an order enters production, it will engage in a relation with the production department. In conjunction with the relations, its capabilities evolve. For example, when it comes into relation with the accounting department, it will get an extra attribute to store the invoice number.

The example given above also serves to illustrate the way relations are modelled using autonomous objects. An order is itself a relation between a client and a product, because it cannot be brought into the system without these objects. This should be taken very literally in an autonomous database. A client object must initiate the order. The relation is then initialised, which means that some conditions on both partners are checked and then the relation objects is created. At the same time the client and product objects involved in the order will be extended, for example the client gets a rule to pay upon arrival of the goods. These are capabilities an object only needs if it is involved in an order relation. Therefore it gains these capabilities when it enters a relation, and loses them again when the relation is terminated.

These characteristics make a number of requirements on the data model. Firstly, objects are autonomous in trying to initiate and terminate relations. Secondly, the capabilities of an object are dependent on the relations it is currently involved in. Therefore we need a mechanism that, often temporarily, extends an object. Furthermore, there must be an existential dependency between objects in the model, if there is such a dependency in the real world. In the next sections we will first explain what an individual autonomous object looks like. After that we will give the classification of data objects that achieves the elements described above, viz. existential dependency and extension of objects.

2.2 An individual autonomous object

To get a more concrete picture of a database of autonomous objects we will first show what an individual autonomous object can do. As stated before, an autonomous object has attributes, methods, rules, and constraints.

Attributes Like any object, an autonomous object is defined by data and behaviour. The data in an object are the attributes defined for it. These can be any basic types, and tuples and sets formed out of these. Attributes are completely encapsulated, there is no access to

them from the outside. Examples of attributes are the maximum overdraft on a bank account or the number in stock of a product object.

Methods Methods define the actions an object is able to perform. Examples are methods to calculate interest, to debit or to credit in a bank account relation object. Another example is a method to schedule production of a new order in a production planning object.

Rules The active behaviour of an object is event-driven. It is defined by event-condition-action (ECA) rules. This means that on a specified event the action is executed if the condition stated is satisfied. All three components are about the object the rule is defined on and on the objects it is related to. A example is a rule in a product object that automatically generates a new order if the inventory falls below a certain threshold.

Constraints The behaviour of an object can be constraint through the definition of dynamic constraints. These are basic process algebraic expressions [5] with guards. This means that dynamic constraints can enforce sequencing. The guards give the possibility to define preconditions on methods. An example of a precondition is that a bank account can only be debited if the maximum overdraft is not exceeded. An example of sequencing is that a book must be checked out of the library before it can be checked in again.

Inheritance Classes can inherit the definition of other classes. There is a limitation on inheritance in that only instances of leaves of the inheritance tree are allowed to exist. This limitation is motivated by its use in the autonomous data model. The capabilities defined on an autonomous object are its inherent, unchangeable capabilities. The inheritance hierarchy in the autonomous data model serves to make it possible to make generalisations of classes. An example is the notion of a legal entity. Both companies and persons are legal entities, but there are no objects that are purely legal entities and nothing else. Specialisation where capabilities are added, is dealt with by the addon mechanism explained in the next subsection.

2.3 Relations between Objects

In a database we have entities such as the bank and natural persons that exist independently. These are the ordinary objects in our database. However, these objects do not lead an isolated existence. In this section we will show how we bring the relations between objects in our model. To illustrate the way relations between autonomous objects are modelled, we first give an example.

Example A bank account is a relation between a person and a bank. A person can open an account with a bank, if he satisfies certain conditions. These conditions vary from presenting a valid identity card to not having a bad credit record. After that the account will be opened, in other words the relation is established. The person now can do a number of things because of this relation. For example, he can transfer money to other bank accounts. Naturally, the relation can also be terminated. Again a number of conditions need to be satisfied. The bank account will not be closed, if there is an overdraft on it. Similarly the conditions of the bank account will describe what happens in exceptional circumstances. An example is that the account is transferred to one of the heirs if the account holder dies.

In an autonomous database we proceed in an analogous way in establishing relations. A relation is defined by a protocol. The protocol describes how a relation is initiated, how it

can be terminated and what is done in exceptional cases. A protocol is a lifecycle definition of a relation. It consistst of dynamic constraints and rules. The constraints assure that the appropriate conditions are satisfied before taking any step in the relationship. The rules trigger appropriate actions for those events that are relevant to the relationship.

The protocol for a relation is stored partly in a relation class object and partly in a relation object. The information needed to initiate a relation is stored in the relation class object. The rest of the protocol, needed during the lifecycle of the object is kept in the relation object.

On an initiation request the relation class object first checks if the objects that request the relation satisfy the conditions in the protocol. If this is successful the actions to actually create the relation are taken. In order to store information about the relation a relation object is created. A relation object is itself an autonomous object. This means that it can at its turn engage in relations with other objects. At the same time the objects that engage in the relation are extended with the capabilities to deal with the relation. This is done through the addon mechanism. An addon defines an extension to an object. If an object is extended through an addon, it acquires the capabilities defined in the addon. Since an addon only defines an extension to an autonomous object, there are no instances of an addon. If an object is extended through an addon, the added capabilities cannot be distinguished from the inherent capabilities.

The relation object accepts request for termination of a relation. Again the conditions in the protocol will be checked and if these are satisfied, the relation is terminated. The actions are basically the reverse of those executed on the creation of the relation. This means that the capabilities added to the objects by the addons belonging to the relation are removed and the relation object ceases to exist. However, the protocol might state application specific actions to be taken. For example, a bank might wish to keep some historical information on closed bank accounts.

Finally, the protocol in the relation objects defines the actions taken in exceptional cases. The most common exception is the situation where one of the partners in the relation ceases to exist. Conceptually the existence of the relation is dependent on the existence of the partners in the relation. Therefore the default protocol will probably be that the relation ceases to exist, if one of the partner objects dies. Again, application specific requirements might lead to another protocol.

2.4 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 [18].

We are not the first in signalling these problems. The concept of an aspect [19] of an object is another solution proposed. However, in addition to extending an object's behaviour an aspect can modify it. An object can be referenced through an aspect. This means that the behaviour of an object depends on the aspect through which it is approached. Hence, an aspect is a kind

of extended view mechanism. It also means that we cannot combine two aspects to extend an object, whereas an autonomous object can be extended by multiple addons at the same time.

Addons offer a clean, Lego-like mechanism to extend the capabilities of an object. The behaviour of an object is extended through the addon mechanism. What the view on an object is from other objects can be defined in the constraint section of a class definition. Thus, access control to, or the view on, an object is separate from the extension of an object's behaviour.

2.5 Structure of the Object Model

The type structure in the autonomous data model is three layered. Object instances have a class as their type and the classes are typed by one of the three metaclasses. The three levels can be described as follows:

Instance Level At this level the instances of (relation) objects live. These objects are the representation of the real world the system tries to model. This means that person objects, bank objects and bank account relation objects are at this level.

Class Level At the class level each class is represented by a class object. The class object takes care of creation and deletion of instances and keeps track of them during their lifetime. Class objects of relation classes have similar tasks as described in the previous section. Addons offering extensions of objects can also be found at this level.

Each object at the class level records the identities of the objects it is involved with. This means that a class object or a relation class object keeps a set with the objects in its class. An addon keeps track of the objects that it has extended. This way all instances of a class can be accessed through the class object. This means that queries for objects of a certain class should be addressed to the class object of that class.

Metaclass Level The metaclasses are also present in the system as objects. The metaclass level is the highest level in the system. The main reason of their presence is to facilitate schema evolution. Through the metaclasses new classes can be added to the database and existing classes can be altered. Analogous to requests for creating and discarding instances accepted by class objects, metaclass objects accept requests to create and discard classes. In the object model described in this section we see three metaclasses, viz. object classes, relation object classes, and object addons.

A sketch of this structure and what lives at each level is given in Figure 2. The arrows indicate which objects at the instance level can be reached from which objects at the class level.

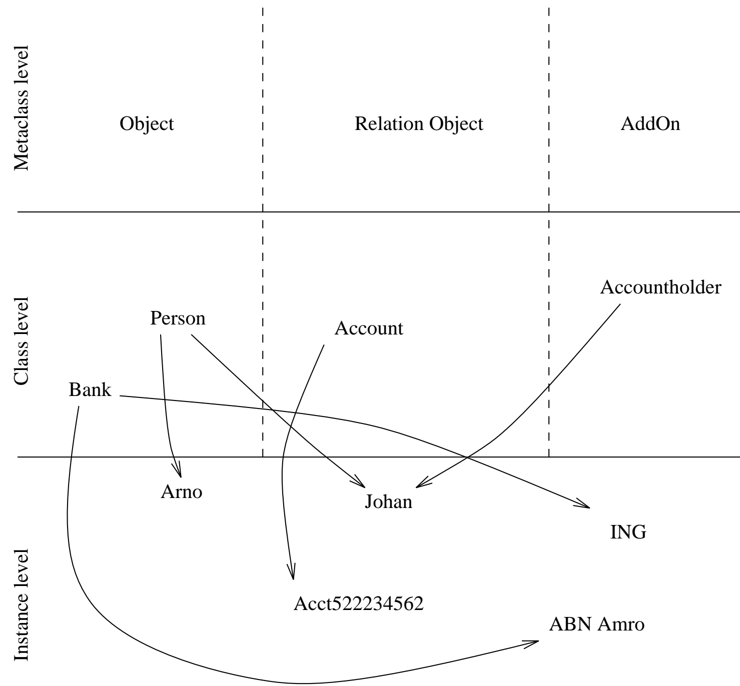

Figure 2: Structure of the object model

3. Syntax and Semantics of Autonomous Objects

3.1 Syntax

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

$$\begin{aligned}\langle \text{Class} \rangle \quad \longrightarrow \quad & \langle \text{ClassHeader} \rangle \\ & \langle \text{AttributeSection} \rangle \\ & \langle \text{MethodSection} \rangle \\ & \langle \text{RuleSection} \rangle \\ & \langle \text{ConstraintSection} \rangle \\ & \langle \text{ClassEnd} \rangle \end{aligned} \tag{3.1}$$

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

A set of basic types	$\langle BasicType \rangle$
A set of functions on the basic types	$\langle BasicFunction \rangle$
A set of constant values for each basic type	$\langle BasicValue \rangle$
A set of Boolean functions on the basic types	$\langle Condition \rangle$
A set of identifiers	$\langle \ldots Name \rangle$
A set of class names	$\langle ClassName \rangle$

The set of basic types includes at least the type *Oid* of object identifiers.

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.

$$\langle\text{Type}\rangle \longrightarrow \langle\text{BasicType}\rangle \mid \mathcal{P}\,\langle\text{Type}\rangle \mid \langle\text{TupleType}\rangle \mid \langle\text{ClassName}\rangle \tag{3.2}$$

$$\langle\text{TupleType}\rangle \longrightarrow (\,\langle\text{FieldList}\rangle\,) \tag{3.3}$$

$$\langle\text{FieldList}\rangle \longrightarrow \langle\text{Field}\rangle \mid \langle\text{Field}\rangle\,,\langle\text{FieldList}\rangle \tag{3.4}$$

$$\langle\text{Field}\rangle \longrightarrow \langle\text{LabelName}\rangle : \langle\text{Type}\rangle \tag{3.5}$$

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.

$$\langle\text{ClassHeader}\rangle \longrightarrow \textbf{Object}\,\langle\text{ClassName}\rangle\ [\textbf{isa}\,\langle\text{ClassName}\rangle] \tag{3.6}$$

$$\langle\text{ClassHeader}\rangle \longrightarrow \textbf{Object}\,\langle\text{ClassName}\rangle\ [\textbf{isa}\,\langle\text{ClassName}\rangle]\ \textbf{Relation}\,\langle\text{ClassList}\rangle \tag{3.7}$$

$$\langle\text{ClassHeader}\rangle \longrightarrow \textbf{AddOn}\,\langle\text{ClassName}\rangle\ [\textbf{isa}\,\langle\text{ClassName}\rangle]\ \textbf{Extends}\,\langle\text{ClassName}\rangle \tag{3.8}$$

$$\langle\text{ClassList}\rangle \longrightarrow \langle\text{ClassName}\rangle\,,\langle\text{ClassList}\rangle \mid \langle\text{ClassName}\rangle \tag{3.9}$$

An example of a class header for an independent object is:

Object Person **isa** LegalEntity

A relation object class BankAccount might have the header:

Object BankAccount
Relation Bank, Person

An addon that extends a Person object to an account holder might have the following header:

AddOn AccountHolder
Extends Person

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

Attributes Declaration of attributes is straightforward. Types possible are basic types, power types and tuple types. Power and tuple types are formed in the usual way.

$$\langle\text{AttributeSection}\rangle \longrightarrow \textbf{Attributes}\ \langle\text{AttributeList}\rangle \tag{3.10}$$

$$\langle\text{AttributeList}\rangle \longrightarrow \langle\text{AttributeDecl}\rangle \tag{3.11}$$

$$\langle\text{AttributeList}\rangle \longrightarrow \langle\text{AttributeDecl}\rangle\ \mathbf{,}\ \langle\text{AttributeList}\rangle \tag{3.12}$$

$$\langle\text{AttributeDecl}\rangle \longrightarrow \langle\text{AttributeName}\rangle : \langle\text{Type}\rangle \tag{3.13}$$

An example attribute section is as follows:

Attributes
number : integer
keywords : $\mathcal{P}$string
position : (x:integer,y:integer)

Methods The methods of an object are defined in the method section of the class declaration. Possible actions in object are assignments to attributes, method calls or set operations. A method call can be either to an internal method or to a method of another object. Set operations can be iteration or selection.

⟨MethodSection⟩	⟶	**Methods**	(3.14)
		⟨MethodList⟩	
⟨MethodList⟩	⟶	⟨MethodDecl⟩ |	(3.15)
		⟨MethodDecl⟩ **,** ⟨MethodList⟩	
⟨MethodDecl⟩	⟶	⟨MethodName⟩ **(** ⟨ParameterList⟩ **)** { ⟨StatementList⟩ }	(3.16)
⟨StatementList⟩	⟶	⟨Statement⟩ |	(3.17)
		⟨Statement⟩ ; ⟨StatementList⟩	
		⟨AttributeName⟩ := ⟨Expression⟩	(3.18)
		⟨MethodCall⟩ |	(3.19)
		⟨AttributeName⟩ := ⟨MethodCall⟩ |	
		⟨SetIteration⟩ |	
		Return ⟨Expression⟩	
⟨Expression⟩	⟶	⟨AttributeName⟩ | ⟨PathExpression⟩ |	(3.20)
		⟨BasicFunction⟩ | ⟨BasicValue⟩	
⟨PathExpression⟩	⟶	⟨Expression⟩ . ⟨Expression⟩	(3.21)
⟨MethodCall⟩	⟶	[⟨ObjectId⟩] . ⟨MethodName⟩ **(** ⟨ActParamList⟩ **)**	(3.22)
⟨SetIteration⟩	⟶	**forall** ⟨VariableName⟩ **in** ⟨AttributeName⟩	(3.23)
		where ⟨Condition⟩	
		do{ ⟨StatementList⟩ }	
⟨ActParamList⟩	⟶	⟨ActParam⟩ **,** ⟨ActParamList⟩	(3.24)
⟨ActParamList⟩	⟶	⟨ActParam⟩	(3.25)
⟨ActParam⟩	⟶	⟨ParameterId⟩ **=** ⟨Expression⟩	(3.26)

An example of a method declaration is the following method that might occur in a bank object. It calculates and adds the interest over all bank accounts the bank object knows.

Methods
CalculateInterest(Rate:real) {
forall Acct **in** RelationSet
where Acct.Type = BankAccount
do {
Acct.add(Acct.calculateInterest(Rate))

```
        }
    }
```

Rules The rule section defines the rules on the object. These are Event-Condition-Action triples as is usual in active database systems.

$$\langle \text{RuleSection} \rangle \longrightarrow \textbf{Rules} \; \langle \text{RuleList} \rangle \tag{3.27}$$

$$\langle \text{RuleList} \rangle \longrightarrow \langle \text{Rule} \rangle \tag{3.28}$$

$$\langle \text{RuleList} \rangle \longrightarrow \langle \text{Rule} \rangle \, , \langle \text{RuleList} \rangle \tag{3.29}$$

$$\langle \text{Rule} \rangle \longrightarrow \textbf{On} \; \langle \text{Event} \rangle \; \textbf{if} \; \langle \text{Condition} \rangle \; \textbf{do} \; \langle \text{Action} \rangle \tag{3.30}$$

$$\langle \text{Event} \rangle \longrightarrow \langle \text{MethodName} \rangle \mid (\langle \text{Event} \rangle + \langle \text{Event} \rangle) \mid (\langle \text{Event} \rangle \, ; \langle \text{Event} \rangle) \mid \langle \text{Event} \rangle * \tag{3.31}$$

$$\langle \text{Action} \rangle \longrightarrow \langle \text{MethodCall} \rangle \tag{3.32}$$

An example of a rule is the following rule that makes a person draw cash with his cashcard when his salary arrives and if his purse has less than £25 in it.

Rules
On monthlySalary
if purse $<$ 25
do CashCard.getCash(75)

Constraints The constraints on the behaviour of an object instance are defined in the constraint section of the class definition. We allow basic process algebraic expressions on the

$$\langle \text{ConstraintSection} \rangle \longrightarrow \textbf{Constraints} \; \langle \text{ConstraintList} \rangle \tag{3.33}$$

$$\langle \text{ConstraintList} \rangle \longrightarrow \langle \text{Constraint} \rangle \tag{3.34}$$

$$\langle \text{ConstraintList} \rangle \longrightarrow \langle \text{Constraint} \rangle \, , \langle \text{ConstraintList} \rangle \tag{3.35}$$

$$\langle \text{Constraint} \rangle \longrightarrow {}'\Big[{}' \langle \text{Condition} \rangle {}'\Big]{}' \langle \text{Event} \rangle \mid \langle \text{Event} \rangle \mid {}'\Big[{}' \langle \text{Condition} \rangle {}'\Big]{}' \langle \text{Event} \rangle \, \langle \text{Constraint} \rangle \tag{3.36}$$

An example of a constraint is the following constraint on a bank account. It defines the dynamics of a facility for telebanking by first requiring a authentification procedure before a series of transfers can take place. The precondition of a transfer is that the balance of the account must be larger than zero.

Constraints
(Login;([Balance> 0] Transfer)*;Logout)*

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. This means that all assignments and method calls are correctly typed.
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.

3.2 Semantics
Class definitions do not define an instance of a database. An actual database is a collection of object instances. Therefore we will concentrate on the individual object in this subsection. The semantics of an autonomous object is that of an ordinary object extended with triggers and constraints on its behaviour. However, the capabilities of an object may vary over time through the evolution of its relations.

3.2.1 Typing The basis of the semantics of an autonomous object are types. Typing of attributes and methods in the autonomous data model follows [7].

Definition 1 *Given a set of basic types B and a set of unique labels L, the set of types T is defined as follows:*

1. *If $\beta \in B$, then $\beta \in T$.*
2. *$(\sigma \rightarrow \tau) \in T$, iff $\sigma, \tau \in T$.*
3. *$\langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle \in T$, iff $m \in I\!N$ and for $1 \leq i \leq m$ $\tau_i \in T$ and $a_i \in L$.*
4. *$\mathcal{P}\tau \in T$, iff $\tau \in T$.*

To this set of types T we add the object types defined in the class hierarchy.

Definition 2 *Given the set of types T and an object hierarchy H, $Type_H$ is T extended with the types induced by H.*

A subtyping relationship is defined on the types following Cardelli [9].

Definition 3 *Let H be a class hierarchy. The subtyping relation $\leq : Type_H \times Type_H$ on H is defined as follows:*

1. *if $C_1, C_2 \in Type_H$ and C_1 **isa** C_2, then $C_1 \leq C_2$.*
2. *Let $\sigma = (\sigma_1 \rightarrow \sigma_2)$ and $\tau = (\tau_1 \rightarrow \tau_2)$. If $\tau_1 \leq \sigma_1$ and $\sigma_2 \leq \tau_2$, then $\sigma \leq \tau$.*
3. *if $\sigma, \tau \in Type_H$ and $\sigma \leq \tau$, then $\mathcal{P}\sigma \leq \mathcal{P}\tau$.*
4. *if $\sigma_1 = \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \in Type_H$ and $\sigma_2 = \langle m_1 : \nu_1, \ldots, m_k : \nu_k \rangle \in Type_H$, 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 and of classes are given in the following definition.

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

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

Definition 5 *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 **isa** C_2, then $D(C_1) \subseteq D(C_2)$.*
2. *If $C_1 \neq C_2$ and not C_1 **isa** C_2, then $D(C_1) \cap D(C_2) = \emptyset$.*

The domains of the types are defined following [7], such that $D(\sigma) \subseteq D(\tau)$ if $\sigma \leq \tau$. We first define the predomains of the types:

Definition 6 *For each type $\tau \in T$ the predomain of τ, $D_p(\tau)$, is defined as follows:*

1. *The predomains $D_p(\beta)$ of a basic type β are postulated in Definition 4.*
2. *$D_p(\sigma \to \tau) = D_p(\sigma) \to D_p(\tau)$, the set of total functions from σ to τ.*
3. *$D_p(\mathcal{P}\tau) = \mathcal{P}D_p(\tau)$*
4. *$D_p(\langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle) = \{\langle l_1 : a_1, \ldots, l_n : a_n\rangle | a_i \in D_p(\tau_i)\}$*

From these predomains we derive the domains as follows:

Definition 7 *For each type $\tau \in T$ the domain $D(\tau)$ is constructed as follows from the predomains:*

1. *For a basic type β, $D(\beta) = D_p(\beta)$*
2. *$D(\sigma \to \tau) = D(\sigma) \to D(\tau)$*
3. *$D(\mathcal{P}\tau) = \mathcal{P}(D(\tau))$*
4. *If $\tau = \langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle$, then $D(\tau) = \bigcup_{\sigma \leq \tau} D_p(\sigma)$.*

Methods are typed as well in the autonomous data model. This is done through function types, like in TM/FM [6]. In this approach a method is a function mapping an object state and input parameters to a new object state and 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*$\rangle$.

We need to be able to express the type of an object at any arbitrary moment in the object's existence. We will call this the current type of an object. The intuition behind the current type of an object is that it is the type induced by the inherent capabilities of an object and the capabilities added by addons.

An object definition, as well as an addon definition, defines a type environment.

Definition 8 *To each object definition and each addon definition D we can apply an operator $Type(D)$ that yields a tuple type $\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$ containing the type environment defined by D.*

Tuple types can be combined by the following tuple type addition operator.

Definition 9 *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 $u_i : \nu_i \in \{t_1 : \tau_1, \ldots, t_n : \tau_n, s_1 : \sigma_1, \ldots, s_m : \sigma_m\}$

Definition 10 *The current type an object O of class C extended with addons $A_1, \ldots, A_m$ is given by:*

$$CurrentType(O) = Type(C) \otimes \bigotimes_{i=1}^{m} Type(A_i)$$

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 [24]. It is used, for example, in TM/FM to correctly type inherited methods. In the autonomous data model the relevant part of the type context for a method M in an object O is given by the 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 $\Delta = Type(Person) \otimes Type(AccountHolder)$.

3.2.2 Object Interpretation The attributes of an object at any moment are the attributes in its class and the attributes added by various addons. The function *Attr* applied to an object instance yields the current attributes of that instance. Since not only the current state is of importance, but also an object's behaviour in the past, every object records its history. It keeps all its historical states and records the method calls that caused the state transitions. In addition each autonomous object knows its own identity.

Definition 12 *The function $Attr(O)$ returns a tuple $\langle \alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n \rangle$ containing all current attributes of object O. Every autonomous object has an attribute with its object identity $self : Oid$.*

There is a similar predicate returning the methods currently defined on an object.

Definition 13 *The function $Meth(O)$ returns a set $\{Meth_1, \ldots, Meth_n\}$ containing all methods object O currently has.*

Each object keeps its history. Both states, i.e. attributes, and transitions between the states, i.e. method calls, are recorded. The history of method calls is also called the event history.

Definition 14 *The history of an object is kept in two elements:*

1. *All historical states are kept.*
2. *Given a set of method identifiers M the function $Seq(M)$ generates all sequences of elements of M. Every autonomous object records its event history, $Hist : Seq(M)$.*

An interpretation for a tuple of attribute declarations assigns a correctly typed value to each attribute.

Definition 15 *For a tuple of attribute declarations $AD = \langle \alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n \rangle$ an interpretation $I(AD)$ assigns values $I(\alpha_i) = k_i$ such that $k_i : \sigma_i$ where $\sigma_i \leq \tau_i$ for $1 \leq i \leq n$.*

An interpretation of an object is an interpretation of the current attributes of an object.

Definition 16 *An interpretation I for an autonomous object O is an interpretation for the current tuple of attribute declarations $Attr(O)$.*

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

Definition 17 *Let $db = \{o_1, \ldots, o_n\}$ be a set of objects. $\Gamma(db) = \bigcup_{i=1}^{n} I(o_i)$ is a pre-model for db*

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

Definition 18 *Given a set of classes C, a pre-model for a database $\Gamma(db)$ is a model for db, iff*

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

From a model of a complete database we can get a model for the part of the database related to a specific object.

Definition 19 $\Gamma(O) \subseteq \Gamma(db)$ *is the model induced by O. $\Gamma(O) = \bigcup_{o \in R_O} I(0)$, where R_O is the set of object reachable through path expressions from O in $\Gamma(db)$.*

3.2.3 Method and Rule Execution The interpretation is the static part of the semantics. The semantics of method and rule execution is part of the dynamic side of the semantics. 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 20 *Given a condition C in object O. C is satisfied by O, denoted by $C(O)$, iff $\Gamma(O) \models C$, where $\models$ denotes the standard logical inference relation [11]. A condition C is typed $C : CurrentType(O) \longrightarrow \{true, false\}$.*

Execution of methods and rules must conform to the dynamic constraints on the object. In addition rules are triggered by the contents of the event history. As we saw above, constraints are regular event expressions interspersed with conditions. Constraints on an autonomous object are similar to the way the dynamics of an object is modelled using finite automata in OMT [20].

Definition 21 *Given an object O, a constraint C is a guarded basic process algebraic expression where the event alphabet is $Meth(O)$ and the guard conditions are conditions as defined in Definition 20.*

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

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 constraint-checking or event-checking automaton in following definitions.*

Proof Follows from the fact that the event expressions are a regular language [17]. The transitions in a constraint-checking automaton are labelled by the preconditions and the method names. A constraint 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 $Hist$ for an event expression. □

An object O has a set of dynamic constraints $DC_O = \{DC_1, \ldots, DC_n\}$ with a constraint checking automaton cca_i associated with each $DC_i \in DC_O$. For example, the finite automaton in Figure 3 implements the following dynamic constraint.

Constraints
(Login;([Balance> 0] Transfer)*;Logout)*

A method is executed if it does not violate the constraints imposed on the object. This means that the object state must satisfy the, possibly empty, precondition given by the constraint. 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.

Definition 22 *A method call M is executed on an object O iff there is a $DC_i \in DC_O$ such that cca_i is in a state with an outgoing transition labelled with a method name m and a condition C, such that $m = M$ and $C(O)$. The method call M is appended to the event history, $Hist := Hist, M$. The method call brings the object on a new state. The previous state is kept with the historical states of the object.*

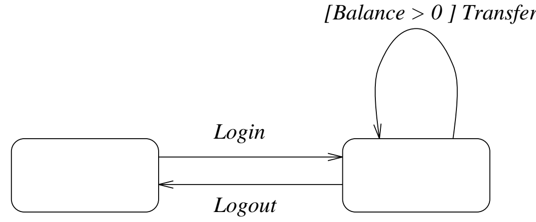

Figure 3: Finite state machine implementing a constraint

The semantics for rule execution is relatively simple because a rule is defined locally on an object. Therefore there is no need to distinguish between set-oriented and instance-oriented semantics for trigger execution [25].

Definition 23 *A rule R on an object O is a triple $\langle E, C, A\rangle$ of an event expression E, a condition C and an action A. E is a basic process algebraic expression on the alphabet given by $Meth(O)$. C is a condition on the state of O as defined in Definition 20. Action A is a method call. If E parses the event history $Hist$ correctly and $C(O)$, then A is executed.*

Above we defined when method execution takes place. We now proceed with the semantics of method execution itself. The semantics of method execution is defined in terms of variant interpretations and the semantics of the basic functions. A variant interpretation relates two interpretations to each other. A variant of an interpretation I is denoted by $I\{v/(a:\tau)\}$. The variant is the same as I, except for $I(a:\tau)$, which returns v. The result of the execution of a single statement is given by a function M:

Definition 24 *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.*

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

Definition 25 *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:*

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

A statement can be applied to all elements of a set simultaneously.

Definition 26 *Let S be a statement, $A = \{a_1, \ldots, a_n\}$ a set, then*

$$M(S, I(A)) = \{M(S, I(a_1)), \ldots, M(S, I(a_n))\}$$

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

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

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 28 *Let* $m = S_1; \ldots; S_k; \textbf{Return } e$ *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*

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

We consider as given the semantics of the basic functions on the basic types. We next give the semantics of assignment.

Definition 29 *Given an interpretation* $I(O)$ *on an object* 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* $M(a_i := m(p_1, \ldots, p_n), I(O)) = I(O)\{a_i = R(m(p_1, \ldots, p_n))\}$.

Set iteration is defined as follows

Definition 30 *A set iteration* SI *in an object* O *is a triple* $\langle A, C_\tau, S \rangle$*, where* $A : \mathcal{P}\tau$ *is a set-valued attribute of* O*,* C_τ *is a condition on a variable of type* τ *and* S *is a statement.*

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

Definition 31 *Given an interpretation* $I(O)$ *on an object* O *the effect of an set iteration* $SI = \langle A, C_\tau, S \rangle$*, denoted by* $M(SI, I(O))$*, is defined as*

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

4. Related Research

Active Databases An autonomous objects has active rules. Therefore a database of autonomous objects is an active database. Some active DBMSs have been constructed using the relational model [16, 25]. Others are based on an object-oriented data model [15, 12]. In most systems there is a separate rulebase, where all rules are stored. In an object-oriented system this rulebase can be integrated with the database [13, 14] in such a way that rules can be treated as objects themselves. A object-oriented system would offer the possibility of defining a rule on an object, a facility present in, for example, SAMOS [15]. Definition of rules is also possible outside a class in SAMOS however. The same is true for Chimera [10].

Chimera integrates deductive, object-oriented and active databases. This means that there is a distinction between passive Datalog rules and active rules, or triggers.

In our model rules are always encapsulated in objects. Because relations are objectified, this does not hinder the definition of rules dependent on multiple objects. If there is a rule that depends on two objects at a time, there must be a relation between those objects on which the rule can be defined. A further important additional feature of our model is the addon mechanism. Both SAMOS and Chimera only offer an inheritance hierarchy. No provision is made for evolution of an object's capabilities during its lifetime.

Concurrent Object Languages The concept of an active object also occurs in concurrent object languages like POOL-T [3] or Procol [8]. Like in the autonomous data model, objects have their own thread of control. Not surprisingly for programming languages there is no means of structuring the data beyond the notion of an object. This makes such languages difficult to use as a basis for a database management system. An example of an effort to do so is PRISMA/DB, a DBMS based on POOL-T [4]. The main difficulty in such a language is to approach the objects, for example to query them. This is problem arises from the fact that nothing keeps track of the objects present in the system. By introducing class objects we have a place where all objects of a class are known.

Because the emphasis is on short-lived programs, object evolution is not usually addressed in concurrent object languages. Contrary to other languages Actor systems [2] do facilitate object evolution in a limited way in the form of replacement behaviour. Means to structure the data are rather scarce however.

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 [21]. The specific tasks of these agents are not very important here. 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.

5. Conclusions and Future Research

In this report we presented a data model based on autonomous objects. On one hand autonomous objects are a logical consequence of encapsulating active rules in the objects in an object oriented database. On the other hand a number of developments in parallelism and networking make systems under central control either infeasible or undesirable, thus necessitating systems built of autonomous components.

The data model proposed in this report gives a framework for structuring data in an active, distributed environment. It does so by founding the model on autonomous objects, objects

whose behaviour includes active rules and dynamic constraints. Relations between objects are objectified. Autonomous objects can be dynamically extended by addons.

The autonomous data model has a three layered structure with an instance level, a class level and a metaclass level. Class objects take care of object creation and deletion, while enforcing the protocol of a class. Furthermore, class objects form the starting point for querying the database. Metaclass objects offers a structure that facilitates schema evolution.

Further research will be the definition of a query model to accompany the autonomous data model. Next we will design and implement a prototype database system based on autonomous objects. After that the design methods to be used with autonomous objects will be our focus of attention. Topics for further study of a more technical nature involve facilities for schema evolution and issues related to the active database aspect of an autonomous database system.

REFERENCES

1. Special issue on intelligent agents. *Communications of the ACM*, 37(7), July 1994.
2. Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, MA, USA, 1986.
3. Pierre America. POOL-T:a parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199-220. MIT Press, Cambridge, MA, USA, 1987.
4. P.M.G. Apers, C.A. van den Berg, P.W.P.J. Grefen, M.L. Kersten, and A.N. Wilschut. PRISMA/DB: a parallel main-memory relational dbms. *IEEE Transactions on Knowledge and Data Engineering*, December 1992.
5. J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1990.
6. Herman Balsters, 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.
7. Herman Balsters and Maarten M. Fokkinga. Subtyping can have a simple semantics. *Theoretical Computer Science*, 87:81-96, 1991.
8. Jan van den Bos and Chris Laffra. PROCOL: A concurrent object-language with protocols, delegation and persistence. *Acta Informatica*, 28:511-538, September 1991.
9. 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.
10. 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.
11. D. van Dalen. *Logic and Structure.* Springer, Berlin, Germany, 2nd edition, 1985.
12. U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

13. 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.

14. 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.

15. 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.

16. 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.

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

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

19. 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.

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

21. Yoav Shoham. Agent-oriented programming. *Aritficial Intelligence*, 60:51-92, 1993.

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

23. 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.

24. 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.

25. J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the 1990 ACM SIGMOD Conference on the Management of Data*, pages 259-270, 1990.

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