DEGAS - An Active, Temporal Database of Autonomous Objects

Johan van den Akker

DEGAS - An Active, Temporal Database of Autonomous Objects

DEGAS: An active, temporal database of autonomous objects.

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor

aan de Universiteit van Amsterdam

op gezag van de Rector Magnificus

Prof. Dr. J.J.M. Franse

ten overstaan van een door het College voor Promoties ingestelde

commissie in het openbaar te verdedigen in de Aula der

Universiteit op maandag 30 maart 1998 te 13.00 uur

door Johannes Fredericus Philippus van den Akker

geboren te Gouda.

Promotor: Prof. Dr M.L. Kersten (CWI/UvA)
Co-promotor: Dr A.P.J.M. Siebes (CWI)

This research was done at CWI (Centrum voor Wiskunde en Informatica), the National Centre for Mathematics and Computer Science in the theme "Data Mining and Knowledge Discovery".

This research was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO).

SIKS Dissertation Series No. 98-1

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

Foreword

Due to their nature, most PhD theses are rather boring stuff to read. I have done my best to turn my thesis into a readable story, but make myself no illusions. As I do very often myself, most people will only read the acknowledgements to see whether they are mentioned. I hope nobody is disappointed.

First of all, I would like to thank Arno Siebes and Martin Kersten for acting as co-promotor and promotor. Arno was a very good advisor during my career at CWI. He stimulated by sharing the visions of the benefits that autonomous objects could bring the world in general, and information systems in particular. He also was a very critical reader of everything I wrote. Martin supplemented this advice by providing the broader perspective. Furthermore, his management of the department created a nice, stimulating environment to do research.

Of my colleagues at CWI, I would like to mention Arjan Pellenkoft, Sunil Choenni, Florian Waas, Jonas Karlsson, Olaf Weber, Annette Bleeker, Peter Grünwald, and Jeroen van Maanen for both scientific and social diversification.

I would like to thank Klaus Dittrich, Peter Apers, Reind van de Riet, Paul Klint, Peter van Emde Boas, and Arnold Smeulders for reading the draft version of this thesis.

Under the name *Schönes Wochenende Bahn* hides a group of people sharing my interest in railways. Of those people, I would like to mention in particular Twan Laan, Marco van Uden, Robert Klein-Douwel, and Kees Smilde. The, by most people's standards, weird habit of going to Germany or further east for the sole purpose of riding trains provided a good escape from academic life.

My reality check was provided by my friends from university, who moved to industry after their Masters'. Known as *Arctic Breeze*, an inheritance from the time we first worked together in a Software Engineering class, the war stories of Michiel Veen, Michel van Maastricht, Michel Knops, Irsan Widarto, Jeroen van Rotterdam and Floris Hack prevented me from slipping into academic isolation.

Of course, I want to thank my mother and my sister. Having finished her PhD thesis before me, Marjan could always support me in the more difficult phases

of my research, having been through them herself. At all important points in my life, I think of my father. I hope he knows and is proud of me.

Finally, CWI turned out to be a lot more than a good place to do research. In my final year at CWI, I met Mariëtte Godin. Some things in life are more important than a PhD thesis...

Amsterdam, January 21, 1998

Contents

1 Introduction 9

I Motivation 13

2 Software of Autonomous Components 15
2.1 Developments in Technology 15
2.2 Developments in Business 16
2.3 Autonomy as a Solution 18
2.4 Defining Autonomy 19
2.5 Conclusion 21

3 Object Autonomy in Context 23
3.1 Active Databases 23
3.2 Temporal Databases 28
3.3 Object-Oriented Databases 30
3.4 The Impact of Object Autonomy 33
3.5 Conclusion 40

II Model 41

4 The DEGAS Object Model 43
4.1 DEGAS Objects 43
4.2 DEGAS by example 47
4.3 Syntax of a DEGAS Object 53
4.4 Querying DEGAS objects 58
4.5 Distribution Model 60
4.6 Conclusion 61

5 Abstract Semantics of DEGAS 63
5.1 Typing 63
5.2 Objects 70
5.3 Method Semantics 71
5.4 Time in an Autonomous Object 73

5.5 Interpretation 76
5.6 Selectors 79
5.7 From pre-history to history 82
5.8 A DEGAS Database 89
5.9 Queries 89
5.10 Conclusion 91

6 Functional Specification of DEGAS **93**
6.1 Preliminaries 93
6.2 Objects 94
6.3 System Layer 107
6.4 Class Objects 111
6.5 Addon Class Object 112
6.6 Relation Objects 115
6.7 Relation Class Objects 117
6.8 Site Objects 118
6.9 Conclusion 121

7 Practical Aspects of DEGAS **123**
7.1 Implementation of DEGAS 123
7.2 Interface to the Outside 136
7.3 Programming in DEGAS 137
7.4 Query Processing in DEGAS 142
7.5 Conclusion 152

III Design **155**

8 Modelling Workflow in DEGAS **157**
8.1 Design Guidelines for DEGAS 158
8.2 Specification of workflow 162
8.3 Designing a workflow in DEGAS 167
8.4 Flexibility of the Workflow 183
8.5 Conclusion 186

9 (Un)decidability Results for DEGAS Objects **187**
9.1 The DEGAS$^-$ Model 188
9.2 Predicates 196
9.3 Decidability Results for DEGAS^{-2} 199
9.4 Decidability Results for DEGAS$^-$ 205
9.5 Conclusion 214

IV Outlook and Conclusions **215**

10 Outlook **217**
10.1 Active Databases and Agents 217

10.2 Ubiquitous Databases 223
10.3 Conclusion 225

11 Conclusion **227**

Bibliography **231**

Nederlandse Samenvatting **242**

Curriculum Vitae **245**

Chapter 1

Introduction

This thesis is about a database of autonomous objects, named DEGAS. In one sentence, DEGAS is a temporal-active object-oriented database language, based on object autonomy. The name DEGAS stands for "Dynamic Entities Get Autonomous Status"[1]. Dynamic entities means data entities extended with the actions on these data. In our discussion, autonomy means two things: complete encapsulation and freedom from central control. Complete encapsulation means a self-contained specification of an object. Freedom from central control means that we can construct a system without a central system element, which is potentially both a bottleneck and a vulnerable part.

The aim of our research was to investigate the potential of a database based on autonomous objects. Hence, our problem statement is:

> *A number of developments lead to databases based on autonomous objects. For such a database, we have the following questions:*
>
> 1. *Is it easy to realise in practice?*
> 2. *Does it facilitate clean, modular application design?*
> 3. *Does it have a simple formalisation?*

Overview

The answer to these questions is given in four parts. These parts discuss the motivation for object autonomy, the DEGAS model for a database of autonomous objects, issues in database design in DEGAS and an outlook to the future of active object systems.

[1] In addition, we were inspired by the name of the Monet main-memory DBMS [Boncz *et al.*, 1996a, Boncz *et al.*, 1996b] developed by Martin Kersten and his team. It also prolongs the tradition of naming database systems after French painters, started by Ingres [Stonebraker, 1986a].

Motivation Part I discusses the motivation for DEGAS. Chapter 2 discuss the developments in information technology and its applications that lead to autonomous objects. On the technological side, the diminishing size and cost of computing units causes an increasing spread of computing power. Since this also means an increasing distribution of information, database management systems must be able to deal with extreme forms of distribution. On the application side, integration of information systems between organisations leads to systems of components with different owners.

Chapter 3 relates autonomous objects in DEGAS to existing research in databases. In particular, we relate it to research in active databases, temporal databases, and object-oriented databases. After a short overview of these areas, we discuss the benefits of DEGAS in these areas. First, it provides a clean modularisation mechanism for active, object-oriented databases. Second, DEGAS provides a unified formalisation of temporal and active database functionality. Third, it incorporates event expressions to specify historical conditions in queries. Fourth and last, DEGAS provides a straightforward object evolution mechanism, that can be used to model roles.

Model Part II gives a specification of DEGAS. Chapter 4 introduces the basic concepts of the DEGAS model. The basis of DEGAS is the object. Objects are related to each other through relation objects, which are fully capable objects themselves. Transient capabilities of objects, such as their roles in relations, are specified by addons. Class and metaclass objects are present in a DEGAS database for object management tasks. To illustrate these basic concepts, we give an example of an application in DEGAS, a stock exchange. We also introduce the syntax of the DEGAS language. Furthermore, DEGAS queries are introduced.

Chapter 5 then gives the formal semantic definition of DEGAS. This definition is given from scratch, starting with the formalisation of the underlying type system. Then, different elements of a DEGAS object are defined, that are integrated to the full DEGAS object semantics. The formalisation of DEGAS is based on a set-based object semantics and process algebra for the dynamic parts. An advantage of using process algebra is the direct translation from DEGAS language syntax to semantics.

The next chapter, Chapter 6, gives a functional specification of a DEGAS database system. It gives the basic actions of each component and their effects. In fact, the complete system, besides a basic communication infrastructure, can be specified as a DEGAS object.

Practical issues regarding DEGAS are discussed in Chapter 7. We discuss the implementation techniques used in a prototype of DEGAS. Furthermore, we show how application semantics can be programmed in DEGAS. This chapter also discusses the practical aspects of DEGAS query processing. It is designed to deal with the specific complications of object autonomy. One of these is the estimation of the quality of a query result.

Design Part III deals with the design applications of applications in DEGAS Design is discussed in two chapters, reflecting two aspects of database design. Chapter 8 discusses the path from an application to a DEGAS object design. The example application is workflow. We show that the DEGAS design principles lead to a design that cleanly separates concerns and that promotes flexibility. Furthermore, the encapsulation of rules in objects and the relation and addon mechanism of DEGAS provide a good basis for the integration of workflow in an active database.

Chapter 9 addresses verification issues. We investigate whether we can decide the termination and confluence of a given active object design. Due to the complexity of this issue, we use a restricted version of the DEGAS model in this chapter. We prove that deciding termination and confluence is only possible for a very basic rule model. This consequently proves the undecidability of these predicates for DEGAS.

Outlook The final part of this thesis provides an outlook to the future of active object databases. This outlook is based on an extrapolation of the developments described in Chapter 2. In Chapter 10, we discuss the interrelation of active object-based systems, like DEGAS, and intelligent agents. We discuss this from two perspectives, viz., the usefulness of agent technology in databases and the problem of data management in a ubiquitous, agent-based, computing environment.

Publications

The following parts of this thesis have been published in other places:

- An early version of the DEGAS data model, discussing its application to computer graphics, was presented at the Eurographics'95 Workshop on Programming Paradigms in Graphics [Akker and Siebes, 1995b].

- A general introduction of the DEGAS data model, focussing on the DEGAS modelling notions, i.e., objects, relations, and addons, was presented at the CAiSE*96 conference [Akker and Siebes, 1996b]. An extended version of this paper was published in the CAiSE*96 special issue of *Information Systems* journal [Akker and Siebes, 1997b].

- A discussion of the interaction between active rules and the historical functionality in DEGAS was presented at the DEXA'96 workshop [Akker and Siebes, 1996c].

- The discussion of the possible application of agent technology in active object database in Chapter 10 was part of the Cooperative Information Agent'97 workshop [Akker and Siebes, 1997c].

- Chapter 8 has been published separately as a CWI Report [Akker and Siebes, 1997a].
- Chapter 9 is based on a CWI Report [Siebes *et al.*, 1995].

Part I

Motivation

Chapter 2

Software of Autonomous Components

As stated in the Chapter 1, the goal of our work on DEGAS is to study a database based on autonomous objects. This chapter discusses the motivation for object autonomy from the viewpoint of software architecture. A number of developments in information technology and its use in organisations promote an architecture based on autonomous components.

Technological developments of interest are mobile computers and distributed processing. The benefit of autonomous components under these circumstances is found in a reduction of overhead. In the use of information technology by organisations, we see a development towards integration and sharing of information between different organisations. Components of these integrated information systems must retain their autonomy for reasons of ownership and control.

We show the benefits of autonomous components, given these current developments. Furthermore, we argue that the object is the most appropriate granularity for studying autonomous components. The chapter concludes with our definition of object autonomy, that is compared to the notion of autonomy in other areas of database research.

2.1 Developments in Technology

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

The development of powerful mobile computers and the spread of wireless communications makes large networks of mobile computers possible [Imielinski and Badrinath, 1994]. A broad variety of information systems will rely on such networks. Examples are information systems to support a large number of sales representatives, or information systems for fleet management of ships or aircraft. In these systems, each mobile node contains data. Ideally, we want to access all data in such a network as one large database. For example, in the information system of an airline, the maintenance department can have access to information about the current status of an aircraft, either to plan maintenance, or to assist the pilot in making decisions in a problematic situation. Although much effort has been put into making databases to inter-operate, it will be very difficult to devise an arrangement 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. Ideally it allows an almost arbitrary collection of objects to be a functioning database. The collection of objects making up the database may vary over time without the need for a system "authority" 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.

Another example is found in massively parallel computing. To further raise the performance of computer systems massive parallelism is seen as a promising road to travel [Bell, 1992]. An important condition for the acceptance of such platforms for general use is the presence of DBMSs. A key problem in such a DBMS is how to distribute data and execution over the available processors. Centralising such decisions poses a large 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 a system with autonomous components.

2.2 Developments in Business

Developments outside the area of computing also promote autonomy of system components. Although there is a movement to increase integration of systems between businesses, either in a chain information system[1], or through a public information infrastructure, nobody wishes to give up control of his part of such a system.

[1] A *chain information system* is an information system serving a complete value chain [Porter, 1985]. A value chain is a group of companies that together form a production process from raw materials to manufactured product. For example, we have a value chain producing automobiles from iron ore, crude oil, rubber, etc. In this process, each output of a production step is worth more than the input. Hence, the producers in this chain are said to *add value* to their input.

Inter-Organisation Information Systems Developments in networking have a number of effects in the way organisations interact. For example, it enables a manufacturer to integrate his information system with his supplier's information systems. 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 control the information seen and updated by outsiders. Although the components are under control of separate owners, we do want to approach such a chain information system as a single (homogeneous) 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 2.1 show such a situation, where Terrific Tyre, Spotless Steel, Cool Car Co., and Dutch Dampers all bring part of their information system into the chain information system, that is depicted by the darker coloured rectangle.

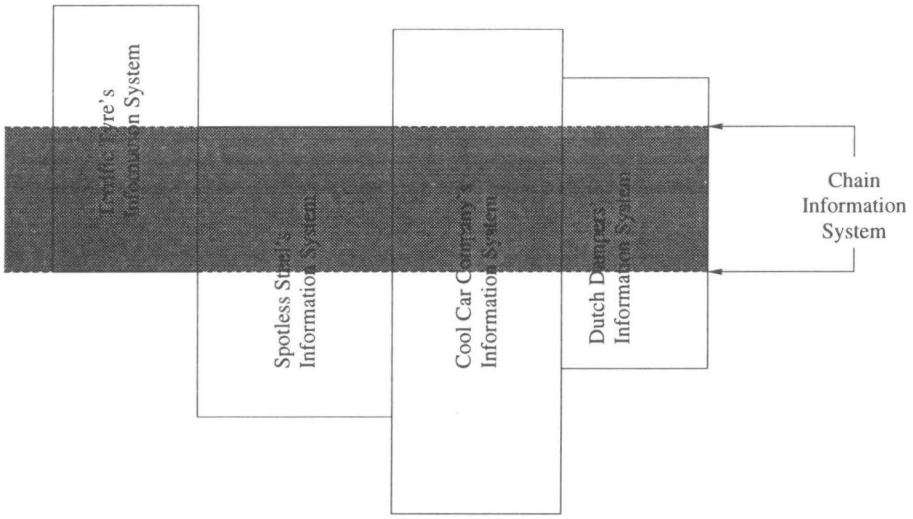


Figure 2.1: The integration of ISs in an inter-organisation IS

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

The problem related to mobile computing, mentioned in the previous section, 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 whether or not to make it (partly) 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, this is a system consisting 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 or Business Process Reengineering [Hammer, 1990, Hammer and Champy, 1993]. 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, see e.g., [Loucopoulos, 1994] and [Glasson *et al.*, 1994], 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 active and independent of other actors. Clearly, it would be beneficial to have a model supporting such autonomous entities in the phases following the modelling phase in the development process.

2.3 Autonomy as a Solution

The developments outlined above lead one to conclude that there is a need for systems composed of autonomous components. Section 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, thus building inherent flexibility into the parts of the system. The result will be autonomy for the components of such a system.

In Section 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 of the components. Exporting data to multiple inter-organisation information systems asks for an inherent flexibility

that autonomous components can offer. It also explained, that current developments in the modelling of organisations tend to emphasise the active and autonomous behaviour of an information system's components.

DEGAS supports 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, since the complexity of the objects may be arbitrary. Hence, 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 object level, naturally fits this model.

Please note, that the motivation for autonomy for component systems is partly similar to the motivation for object autonomy, as discussed earlier in this chapter. [Garcia-Molina and Kogan, 1988] states 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, that the failure of one object affects other objects as little as possible.

2.4 Defining Autonomy

Above, we discussed the benefits of object autonomy in light of recent developments in technology and business. In this section, we give our definition of object autonomy. It will serve as the key foundation of the work on DEGAS, presented in this thesis. After our definition of object autonomy, we compare it to other types of autonomy in databases

Definition 1 *We define object autonomy as follows:*

> *Object autonomy is the maximal encapsulation of specification, control, and execution of an object.*

This definition gives the core notion of object autonomy. Its practical consequences in the DEGAS model are formulated by the following principles:

1. **Every object has a separate thread of execution.** All objects run in parallel. This is encapsulation of execution.

2. **Complete encapsulation of the behaviour of an object.** 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 an element of encapsulation of specification.

3. **Strictly regulated access to an object.** A DEGAS object specifies exactly what objects have access to its actions. Relations between objects specify exactly what data is shared, and what actions can be accessed. This is an element of encapsulation of specification.

4. **Minimal guarantees about an object's behaviour to other objects.** A DEGAS object guarantees as little as possible about its behaviour to other objects. If it gives guarantees, these are specified explicitly. This is an element of encapsulation of control.

5. **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 an element of encapsulation of control.

6. **Autonomy must be given up explicitly.** A further guiding principle is, that if an object gives up autonomy, then this must be explicitly specified.

These principles guide the development and use of DEGAS, as discussed in this thesis.

The notion of autonomy is also encountered in the area of distributed and federated database systems [Sheth and Larson, 1990]. Research in federated databases 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, 1990] distinguish four dimensions of autonomy for a component database of a database federation. These are:

1. **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.

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

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

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

Here, the first two dimensions are about the autonomy of the designer, while the other two dimensions are about the autonomy of the system. Other criteria for the autonomy of the nodes in a distributed database system were formulated in [Garcia-Molina and Kogan, 1988]. They distinguish the following dimension of autonomy:

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

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

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

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

Again, the first two dimensions are about designer autonomy, and the other two dimensions about system autonomy. 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.

The applicability of Sheth and Larson's autonomy criteria for component systems to object autonomy is limited in some respects. The aim of DEGAS, formulated in Chapter 1, is to study the impact of fine-grained autonomy in databases. Hence, not all criteria for autonomy of component databases are useful for the definition of object autonomy. Sheth and Larson's design autonomy for an object means only defining a communication interface between objects, as is done by CORBA [OMG, 1996]. Their communication autonomy means for an object, 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 is fully applicable to an object, since it implies that an object decides internally, what actions to execute, and when to execute them. Likewise, the association autonomy of an object means the freedom to decide the visibility of its parts to other objects.

2.5 Conclusion

In this chapter, we presented the motivation for autonomy in software. The underlying developments are increasing distribution and mobility of information systems, and the increasing integration of information with different owners. We discussed why software of autonomous components is needed under these circumstances. For the study of autonomy in software, we take an object as the granularity. Since objects can be loaded with arbitrary functionality, results applicable to objects are applicable to any type of module.

This chapter closed with the presentation of object autonomy in DEGAS. Object autonomy in DEGAS means maximal encapsulation. We discussed the impact of

this definition on a number of object dimensions, viz., execution, specification, and control. A comparison of object autonomy in DEGAS with autonomy in federated databases showed a great degree of commonality. A difference is caused by the different aims, since DEGAS is also concerned with the internals of the objects in a system.

Chapter 3

Object Autonomy in Context

The previous chapter discussed the developments motivating object autonomy. It also gave the definition of object autonomy used in the development of DEGAS. This chapter puts the concept of object autonomy in the context of databases. DEGAS mainly builds on work in active databases. Hence, we open this chapter with a short overview of this area. Since object autonomy opens new perspectives on the link of active databases with temporal databases, and with object-oriented databases, we also give a short introduction to these two areas.

The second half of this chapter addresses the opportunities of object autonomy in active databases. The maximal encapsulation applied to rules leads to a clean modularisation of the active database. Furthermore, the view of an autonomous object as a process allows a single model formalising a database that is both active and historical. Another innovation is the use of event specifications for temporal queries. Finally, the DEGAS object model reflects the maximal encapsulation of object autonomy. The object specialisation mechanism is very simple and flexible, so that an object has the capabilities it needs, when needed.

3.1 Active Databases

This section gives a short introduction to active databases. First, we give a brief historical overview. Then, we discuss the core area of active databases, viz., rule specification and execution. Readers interested in a more elaborate overview of active databases are advised to read [Widom and Ceri, 1995].

3.1.1 History

The term *active databases* was coined in [Morgenstern, 1983]. There, the term was used to denote a database management system that automatically updates views and derives dependent data. Hence, this proposal did not introduce a

separate notion of rules or triggers. These were first introduced as a separate element in a database management system in [Stonebraker, 1986b].

The original goal of rules was to provide a more flexible mechanism for constraint enforcement. Without rules, the only possible action in case of a constraint violation was to abort the offending transaction. In many cases, however, more constructive actions exist. For example, a circuit design application often has the constraint that a minimum distance between wires must be maintained. If a new wire violating this constraint is inserted, a possibility to correct this is to shift it, such that the minimum distance is maintained. Clearly, it would be beneficial if the design database executes this action without user intervention.

As pointed out above, another early application of active rules is derived data. Like for constraint maintenance, the main advantage of active rules for derived data is increased flexibility. Active rules allow the database designer to choose the time of derivation, e.g., either on insertion of an underlying value, or on retrieval of the derived value.

With the advent of object databases, the incorporation in the database of behavioural elements of the application became accepted. Hence, it was found that large parts of an information system can be encoded by active rules. In particular, parts of applications that reflect an organisation's common business practices can be implemented by active rules. In the world of information systems modelling, these are known as the *business rules* of the organisation [Herbst *et al.*, 1994].

In recent years, a number of prototype active DBMSs have been developed. The most important systems are HiPAC [Dayal *et al.*, 1988a], Starburst [Widom, 1996], SAMOS [Gatziu *et al.*, 1991] and Chimera [Ceri *et al.*, 1996]. Of these systems, Starburst is based on an extended relational database. HiPAC and SAMOS have an object-based data model. Chimera's data model is an object data model, but is heavily influenced by deductive databases. In addition, a number of commercial systems, such as Oracle, Sybase and Ingres, includes a trigger facility. Furthermore, active rules are part of the SQL3 standard [ISO, 1994], which is being formulated.

The use of active databases was classified in [Kappel and Schrefl, 1996]. It categorises applications of rules as follows:

1. maintaining static integrity constraints
2. maintaining derived data and materialised views
3. maintaining dynamic integrity constraints
4. database access authorisation
5. work step ordering

6. representing permissions to act

7. representing obligations to act

Of these seven, the first four are implementations of DBMS functionality. The fifth is relatively specific to certain applications, in particular workflow management applications. It can be regarded as a special case of the third application. The last two applications are forms of business rules. Business rules are a specification of company policy, or a description of the behaviour of a company. Simple examples are rules in an inventory application, that reorder an item if the stock falls below a certain level. More advanced business rules describes the competence of persons in the organisation. These business rules are the rules used to specify application functionality.

3.1.2 Rule Specification

The generally used format for rules in active databases is the Event - Condition - Action (*ECA*) format. The informal meaning is, that on occurrence of event E, if condition C is satisfied by the database, then action A is executed.

Event Event specification is based on a set of basic events and a set of event operators. In general, the set of basic events consists of three categories: database events, temporal events and external events. The set of database events depends on the data model the active DBMS is based on. For a relational database, the usual database events are `Insert`, `Delete` and `Update`. These denote, respectively, the insertion, deletion and update of a tuple in a relation. Further database events are related to transaction processing, such as `commit` or `abort`. An object-based database offers the same database events, with objects instead of tuples as the entities involved for `Insert`, `Delete` and `Update`. Since an object based system adds behaviour to a database through methods, object-based active database systems usually also include method calls as database events.

Temporal events are used to relate database activities to a clock. Absolute temporal events, e.g. `11 FEB 1997 20:00:00`, refer to one specific point in time. Periodic events, e.g. `every MON 05:00:00`, can be used to schedule repeating activities. Finally, relative temporal events specify a duration relative to another event. An example in an object-based database is `2 weeks after Confirm`, where `Confirm` is a method of a database object `Order`. In a relational database, an equivalent specification would be `2 weeks after ConfirmedOrders.Insert`, where `ConfirmedOrders` is the relation containing the confirmed orders.

External events are used for communication with the outside world. Two possible sources of external events are interaction with the user or messages from other applications. An example is a `high-temperature` message from a temperature sensor in a control application.

Basic events can be combined to form composite event expressions through the use of event operators. The most common event operators are sequential composition and alternative composition. Some active DBMSs offer a large range of event operators, yielding a complex event language. An example is ODE [Gehani *et al.*, 1992]. Studies of active database applications [Appelrath *et al.*, 1996], however, have shown that relatively simple rule languages, i.e., rule languages with alternative and sequential composition of events, are sufficient for most applications.

Condition The condition of a rule is a predicate on the database. If the predicate is true, then the condition of the rule is satisfied. A number of systems allow the condition to refer to the state before and after the triggering event. An example is the following rule, that is triggered by salary increases of more than 25 percent:

On Update(Emp.Salary)
if Salary > 1.25 * **old** Salary
...

In this example, `Salary` qualified by `old` refers to the value of `Salary` before the `Update` action.

Action In most active database systems, the action can be an arbitrary database action. For example, it can include transactional commands, such as a `rollback` instruction. In relational systems, we can add any relational retrieval or update action. In object-oriented systems, method calls can be made in a rule's action.

3.1.3 Rule Execution

The semantics of rule execution can vary on a number of dimensions. These are the time and granularity of rule checking, the selection of rules to execute, on which objects to execute, and the coupling between the three parts of a rule.

The first dimension of rule execution is, when the rule engine checks for triggered rules. A natural point to do this is at the end of a transaction, since it would be very difficult to enforce constraints for updates of smaller granularity. The check generally considers the *net effect* of the transaction. For example, if a tuple is inserted and then updated, the net effect is the insertion of the updated tuple. Likewise, if a tuple is inserted and deleted later on in the transaction, the net effect is empty. In object-oriented databases, another natural moment to check rule triggering is after a method call. In addition, an active DBMS might provide a primitive to force a rule checking. An example is the `process rules` command in Starburst [Widom, 1996].

A further dimension of rule execution is how rules execute on the objects, or tuples, they are triggered on. In the literature [Widom *et al.*, 1991], *set-oriented*

and *instance-oriented* semantics are distinguished. Under the former, a triggered rule executes simultaneously on all objects that satisfy its condition. Under the latter, the triggered rule executes, non-deterministically, on one object that satisfies this condition at a time. As we will discuss in Chapter 9, this affects the result of rule execution. The execution sequence of triggered rules can be influenced by introducing priorities between rules. High priority rules are executed before low priority rules. The last choice in the selection of rules to execute, is whether a system executes all triggered rules, or picks only a single rule for execution.

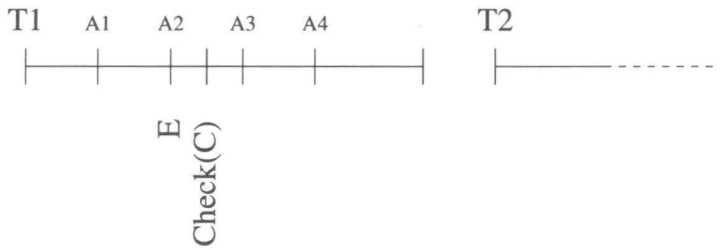


Figure 3.1: Immediate coupling of event and condition

Since rules consist of three parts, another dimension of rule execution is the interaction of these parts. This is known as the *coupling* of these three parts. Two couplings are of importance: Event - Condition coupling and Condition - Action coupling. Three main coupling modes are distinguished: immediate, deferred and independent. These modes assume a flat, i.e., non-nested, transaction model as used in most DBMSs. An immediate coupling between event and condition means that the condition is checked immediately on occurrence of the event, in the same transaction. For example, in Figure 3.1, condition C is checked immediately after the event E occurred in transaction $T1$.

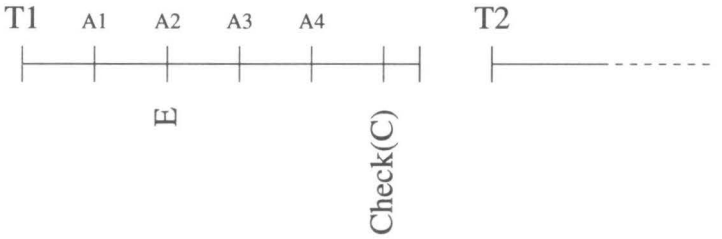


Figure 3.2: Deferred coupling of event and condition

With deferred coupling the condition is checked in the same transaction as the event occurred, but only at the end of the transaction, as depicted in Figure 3.2.

Finally, independent coupling means that the condition is checked in a separate transaction, as depicted in Figure 3.3. With a more advanced transaction model, e.g., the nested transaction model of HiPAC [Dayal *et al.*, 1988a], additional coupling modes are possible. An example is the causally-dependent decoupled mode in HiPAC. In this mode, the rule's action is executed in a separate subtransaction, that can commit only if the triggering transaction commits.

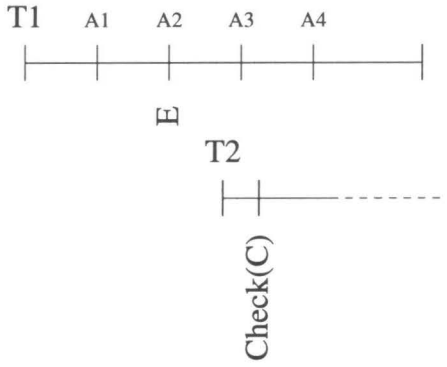


Figure 3.3: Independent coupling of event and condition

3.2 Temporal Databases

An information system reflects the state of a part of the real world, that is subject to change. In many applications, we need facilities to consult data from the past. For example, a bank wants insight in the amount of money flowing into and out of your bank account during the past year to assess your creditworthiness. This need to store data in relation to time is supported by temporal databases. This is a considerable extension of DBMS functionality, due to the complexity of temporal data. This complexity is mainly found in the many possible data models, different temporal dimensions, and interval operators. In this section, we give a short introduction of temporal databases. The reader is referred to [Tansel *et al.*, 1993] for an elaborate overview of temporal databases.

Temporal Dimensions If we have data to be entered in a temporal database, different criteria can be applied to give time stamps. One criterion is the time the data was entered into the database. This is called *transaction time.* Another criterion is the validity of the data in the real world, which is known as *valid time.* For example, in the Netherlands, a new born baby can be registered with the Registry Office up to two working days after birth. Suppose a baby is born on Sunday, April 14th, 1997 and registered on Tuesday, April 16th, 1997. Then in the Civic Registry, April 14th, 1997 would be the valid time and April 16th,

1997 the transaction time of the birth. This example shows a drawback of using transaction time only, viz., the potential lag between the time of validity and the time of entry. Using valid time only, however, also has its disadvantage. If data is entered incorrectly, the previously stored data is lost in a valid time database. Although we can represent past states of the world with only valid time, we cannot reconstruct past database states. Therefore, a full temporal data model, e.g., the data model underlying TSQL2 [Snodgrass, 1994], incorporates both transaction and valid time. Databases containing only transaction time allow us to reconstruct past database states. Usually, these are called *historical databases.*

Data Models 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. This is known as *tuple time-stamping.* Another approach, known as *attribute time-stamping,* is to record the time of validity per attribute. An overview of temporal relational algebras, and the design decisions in defining them, is given in [McKenzie and Snodgrass, 1991].

Object-based temporal data models are less common. In [Wuu and Dayal, 1992], 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 [Ginsburg, 1993]. 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 in [Sciore, 1991]. In an object-based temporal model, we can either have interval time-stamps or point time-stamps. In the former case, a time-stamp gives the complete interval of validity. In the latter case, a time-stamp gives the starting point of a value's validity. Given a number of valuations, we can then infer the interval. Due to the lack of object identifiers, this is not possible in a relational model.

Temporal Queries Querying a temporal database is more complex than querying a database without a temporal dimension, because of operations on time intervals. The temporal dimension influences queries in a number of different ways.

In a temporal database, the result of a query can be a time interval. An example of this is the query "When was the price of Philips shares higher than 80 guilders?". More complex queries for time intervals involve comparisons between intervals. As an example, consider the query "Give the interval when Philips shares were more than 80 guilders and IBM shares were more than 125 dollars". This query asks for the intersection between two intervals, viz., the interval when Philips' share price was higher than 80 guilders and the interval when IBM's share price was higher than 125 dollars. Besides intersection, the union of two time intervals is useful to support queries for time intervals.

A query's condition can include conditions on time intervals. An example is the query "Find the salary of Jones when Smith was his manager." Here, we have a condition on that two intervals must overlap. The first interval is the validity of the returned value for the salary. The second interval is the validity of the manager attribute having value Smith. This type of queries is supported by predicates such as BEFORE, DURING, OVERLAP, et cetera. These represent the usual boolean functions on intervals in general.

In addition to these interval operations, a number of aggregate functions also have temporal variants. Time can be included through the application of aggregates to intervals. For example, a temporal `sum` operator can be used to find the total duration of a condition's truth. A different type of function takes an interval as input parameter in order to find an aggregate value for the given interval. For example, functions like `max`, `min`, and `average` can be applied to intervals. In this case, the operator `max` yields the maximum value of an attribute within the specified interval.

More purely temporal operations in queries are restriction of a query's temporal scope and queries to find specific intervals. The first operation is also known as time-slicing. A time-slice restricts the result to part of the database history. For example, a time-slice [1992, 1995] only yields results between January 1st, 1992 and December 31st, 1995. Conceptually, this is just another temporal predicate. For clarity, it is often put into a separate clause. The second operation, finding intervals, finds smallest or largest intervals satisfying a specified condition. An example is a query "What is the largest interval during which the price of Philips did not exceed fifty guilders?"

3.3 Object-Oriented Databases

Object-orientation is the combination of data and behaviour in objects that have a close correspondence to real-world objects. It was first found in SIMULA [Birtwistle *et al.*, 1974] to structure computer programs for simulations. It was taken further by languages such as Smalltalk [Goldberg and Robson, 1983], C++ [Stroustrup, 1991], and Eiffel [Meyer, 1988].

Important for the introduction of object-orientation in databases were the systems O_2 [Deux, 1990] and GemStone [Maier and Stein, 1987]. In this section, we give a short overview of object-oriented databases. For a general discussion of the object model in databases the reader is referred to [Kim, 1995]. The Story of O_2 [Bancilhon *et al.*, 1992] gives a good overview of issues in building an object-oriented database management system.

The first occurrence of object-oriented notions in databases is in the Entity-Relationship model [Chen, 1976]. Until the mid-eighties, object-orientation was only found in data modelling. Implementation of an information system was

mainly done using relational databases. The broadening scope of information systems brought to light a number of shortcomings of relational systems for advanced applications, like design databases, manufacturing databases, and office automation systems. In such applications, an object-oriented database offers better facilities to model complex structures. A significant advantage of object-oriented databases is the encapsulation of operations with the data. This way, operations shared between programs are specified and stored in a single place. For example, most applications using a manufacturing database need to obtain the composing parts of an assembly. In this case, it has obvious advantages to specify a single operation for this together with the data specification of an assembly. Furthermore, object-orientation offers better facilities to view data at different abstraction levels. For example, we might want to view an aircraft design at the level of the complete aircraft, split up into wings, fuselage, engines etc., or completely "exploded" into parts.

Research in object-oriented databases has not yet yielded a single, well-defined data model, as was achieved very early for the relational model [Codd, 1970]. Hence, standardisation has been actively pursued by both the industrial and the academic parts of the OODBMS community. This effort resulted in the ODMG (Object Data Management Group) model [Catell, 1994]. ODMG defines an interface to and a data model for an OODBMS to promote portability of applications between DBMSs.

This absence of a single well-defined data model led to the formulation of an OODBMS' key properties in the often cited object-oriented database manifesto [Atkinson *et al.*, 1989]. These are:

1. **Complex Objects.** A complex object is an object built from simpler ones. An example is a `car` object that exists of other objects, viz., `part` objects. Complex objects can also be recursive. For example, in a design database, a `subassembly` object can consist of other `subassembly` objects. This construction of complex objects from other objects is called *aggregation.*

2. **Object Identity.** Object data models are based on identity, as opposed to the relational model, which is value-based. In the relational model, two tuples are the same, if their attribute values are equal. In an object-oriented data model, two objects are the same if and only if their identities are the same.

3. **Encapsulation.** This has two aspects. The first aspect originates in abstract data types. It is concerned with the separation of interface and implementation. Additionally, this allows us to hide private data of an object. The second aspect is the combined specification of data and behaviour in an object. This is the important aspect from a database point of view.

4. **Types and Classes.** Two key notions in an OODBMS are types and classes. In an object-oriented system, a type is a specification of an object's features. The type of an object is often given as a tuple of attribute and method types. A class is a collection of objects, that is used to create and store objects.

   Usually, the notions of class and type are closely associated. The relation between types and classes can be in two directions. Commonly, objects in a class conform to the associated type, because they are instances of that class. Another approach is that objects belong to a class, because they conform to the associated type. This is the case in data models allowing arbitrary addition and deletion of attributes, methods and other elements of objects, such as Self [Ungar and Smith, 1987] and Goblin [Kersten, 1991].

5. **Class or Type Hierarchies.** Classes and types are part of hierarchies, formed by *inheritance*. A *subclass* inherits the features from a *superclass*, which means that it has the same data and behaviour, possibly extended with its own data and behaviour. Hence, the subclass is a *specialisation* of the superclass. Likewise, a superclass is a *generalisation* of its subclasses. Everywhere an object of the superclass is required, an object of the subclass can be used.

6. **Overriding, Overloading, and Late Binding.** With the separation of interface and implementations, subclasses of a superclass might have the interface of an operation in common, but have a different implementation. A classical example is a `display` operation for a `graphic` object, which is implemented differently by its subclasses `circle`, `triangle`, and `polygon` objects. Since the name `display` denotes different operations, it is said to be *overloaded*. If the `graphic` object implements its own, generic, `display` operation, then the subclasses are said to *override* this operation with their own definition.

   Overloading and overriding operations means choosing an implementation to execute for each invocation of the operation. For example, if we invoke the `display` operation on a `graphic` object, then we would like the system to execute the most-specific implementation, e.g., the `triangle` implementation of `display` for `triangle` object. This is achieved by *late binding*, which means that the implementation is chosen at the actual execution time.

7. **Computational Completeness.** An OODBMS must allow every computable function to be computed in its data manipulation language.

8. **Extensibility.** The user of the OODBMS must be able to define his own types. Furthermore, there is no distinction in use between system-defined types and user-defined types.

9. **An OODBMS is a DBMS.** An OODBMS must support persistence, secondary storage management, concurrency, recovery, and an ad-hoc query facility.

The final requirement is stated as five separate requirements in the Manifesto itself. The other requirements are on the data model, that also apply to object-oriented programming languages. Actually, this close relation to programming languages is one of the main advantages of an OODBMS over a relational DBMS for complex applications. Most programming languages use a data model, that is different from the relational model. In particular, many programming languages do not have a set-construct, while results from relational queries are always sets. Hence, we have an *impedance mismatch* between the programming language and the relational DBMS. Since an OODBMS uses the same data model as an OO programming language, the impedance mismatch is solved here.

3.4 The Impact of Object Autonomy

Autonomous objects build on the field of active databases. In this section, we discuss the impact of object autonomy on an active database. Secondary themes in the research are the links between active databases, and temporal and object databases. Hence, the scope is the upper, darker coloured, triangle in Figure 3.4.

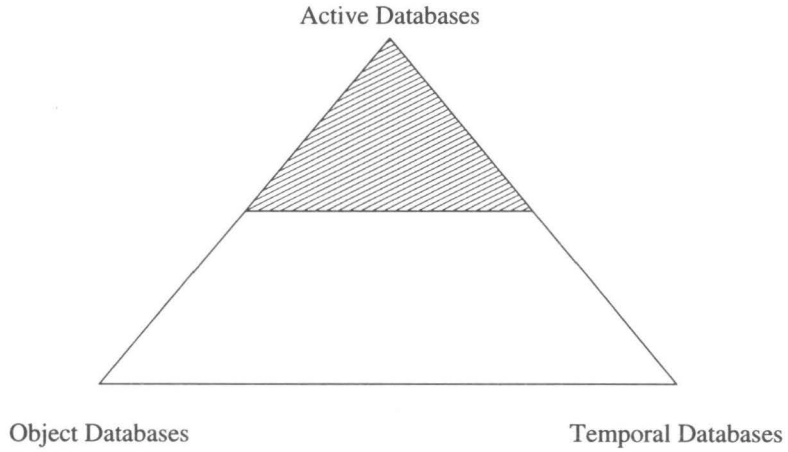


Figure 3.4: The scope of the research in this thesis

We will discuss the impact of object autonomy on an active database in the following four issues:

1. Modularisation of an active database.

2. A unified, process-algebraic, formalisation of active and historical databases.
3. Queries in an active database.
4. Object evolution.

In these four areas the maximal encapsulation of object autonomy offers significant advantages, which we discuss below.

3.4.1 Modularisation of Active Databases

The consequence of object autonomy for the modularisation of an active database is the encapsulation of rules in objects. DEGAS is the first active data model to consequently apply this object-oriented principle to an active database. Other active object databases, like Chimera [Ceri *et al.*, 1996] and SAMOS [Gatziu *et al.*, 1991], offer a hybrid rule model. Rules can be encapsulated, but the separate definition of rules is still allowed.

The modularisation of rules is important, if we have a large number of rules in a database. This is the case if we use rules to implement large parts of an application's functionality. Clearly, in such a situation a single, flat rulebase does not promote easy maintenance and understandability of the system. Hence, an active database needs facilities to bring structure to the rulebase. Since the data in a database is already structured by some means, i.e., through relations in a relational database or through objects in an object-oriented database, rules can either have a structure separate from the data, or have the same structure as the data.

Separate structure for rules and data. If rules use a separate structure, the rulebase is separate from the database, as depicted in Figure 3.5. Here, the modularisation applied to the rules is different from that applied to the data. In some systems with a separate rulebase, such as Starburst [Widom, 1996], rules are grouped in rule sets. One of the main uses of these rule sets is to activate and deactivate several rules at a time. Consequently, the main criterion for modularisation is functional.

A number of different criteria can be used for grouping rules in sets. [Baralis *et al.*, 1996] gives three different criteria, using the term stratification[1]. Behavioural stratification groups rules that together perform a given task. Assertional stratification groups rules that progressively establish some assertion on the database, which is the post-condition of the stratum. Event-based specification groups rules that share a set of triggering events, or that share a set of produced events.

[1]The use of this notion from deductive databases is explained by the strong influence of that area on the work of [Baralis *et al.*, 1996]

Another example of separate modularisation for rules is HiPAC [Dayal *et al.*, 1988b]. In HiPAC, rules are treated as objects. Consequently, the rulebase is itself an object database. According to [Dayal *et al.*, 1988b], the advantage is the availability of the mechanisms for manipulating data for manipulating rules. For example, rules can be part of an inheritance hierarchy. Furthermore, they are subject to transaction control, like any other item in the database. The structure of the rulebase, however, is unrelated to the structure of the database.

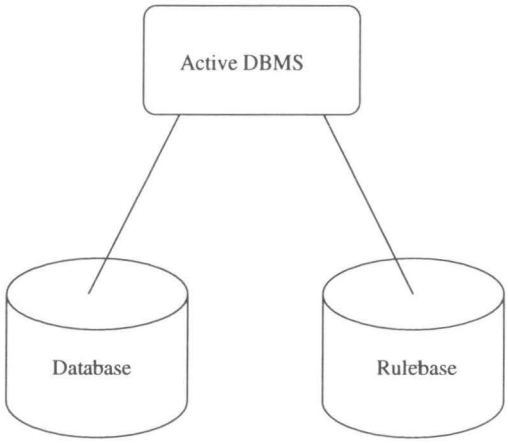


Figure 3.5: Separation of Database and Rulebase

Same structure for rules and data. The other approach is that rules follow the same modularisation as data in the database. For a relational system, this means that rules are defined on the relations in the database. This approach is followed by the SQL3 rules standard [ISO, 1994]. In object systems, the data is modularised according to a class hierarchy. If rules follow the object structure, objects also encapsulate rules. An example of an active object database that provides encapsulated rules is Chimera [Ceri *et al.*, 1996]. The modularisation in Chimera, however, is hybrid, since it still allowed to define rules separately.

In DEGAS, we opt for the complete encapsulation of rules in objects. Hence, the modularisation of the data is applied to the rules without any changes. This limits the number of concepts used in the database design. Furthermore, since rules are part of the behaviour of the data, it is a rigorous application of object-oriented principles. This encapsulation of rules also has the advantage of having all aspects of an object's behaviour defined in one place. This independence of an object's specification is a necessary consequence of object autonomy.

An older system that aims to integrate active rules into objects is MOKUM [Riet,

1989]. 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.

3.4.2 Formalisation of Active and Historical Databases

Object autonomy has a distinct impact on the formalisation of a database. Since one consequence of object autonomy is that every object executes as a separate process, the execution of a single process is the basis of the formalisation of DEGAS. This process-centered view motivates the choice for process algebra, in particular ACP [Baeten and Weijland, 1990], as a central element in the formalisation, as shown in Chapter 5. The key notion in ACP is to match a process specification against the trace of an executed process. If we apply this to an autonomous object, the trace is the history of method execution and process specifications are event specifications. This clearly indicates a link between the history of the database and its rule facility.

The inherent temporal element in active databases was also observed by [Dittrich and Gatziu, 1993] and [Widom and Ceri, 1995]. This temporal element is caused by the inclusion in ECA rules of event expressions composed of multiple basic events, such as method calls [Dayal *et al.*, 1988b, Gatziu *et al.*, 1991], and time events [Dayal *et al.*, 1988a, Hanson, 1989, Gatziu *et al.*, 1991]. This also shows that an event specification is a condition on the history of the database.

We can also see this through a more detailed look into rule triggering. In order to detect complex events, we need to store the basic events occurring in the database. Since a complex event expression usually specifies a sequence of events, the record of basic events must store information about the order in which events occurred, e.g., in an event queue.

This inherent temporal element in active databases raises the question of the relation to temporal and historical databases. 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". In addition to explicit time events, it is desirable to refer to an event's time of occurrence. One possible approach is that the active database gives

access to the time of occurrence in the condition, either through a specific operator [Gatziu and Dittrich, 1993], or by allowing a time parameter to be bound to each event, as is done in DEGAS. Another approach is to specify the temporal conditions on the event by putting the appropriate time events in the event specification. This choice makes 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 case, the time events are included in the event detection mechanism.

Since most active database management systems offer the possibility to specify parameters of events, they 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.

It should be clear by now, that every extension of event specification in the definition of rules beyond single basic events necessitates a partial record of the database history. In particular, if an active DBMS offers all facilities described above, it has to store all method calls with their parameters and timestamp. Obviously, 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. Since DEGAS aims to offer full active database functionality, the state of a DEGAS object includes its history, i.e., a record of past states and method calls. DEGAS offers a single temporal dimension, viz., transaction time, to retain a simple active database model.

Earlier research into the common ground of active and temporal databases mainly focussed on temporal conditions in rules [Gal *et al.*, 1996, Sistla and Wolfson, 1995]. This work extended the condition of a condition - action rule, allowing the specification of an attribute's change over time. An example is to trigger, if the salary of an employee doubles within a year. Since this work did not involve events, it is limited from the standpoint of active databases. Hence, an innovation in DEGAS is the inclusion of temporal functionality in an active database offering full ECA rules.

The formalisation of DEGAS in process algebra has a further advantage. This is found in the direct formulation of the semantics. In active database systems, events are defined and specified in varied ways. Although the complexity of event algebras varies, the algebra can usually be reduced to a small set of operators [Gehani *et al.*, 1992]. Hence, the main difference of interest is in the formalisation of the event algebras.

Two main approaches can be distinguished here. One approach is to translate the event specification to another formalism with a well-defined semantics. For example, in SAMOS [Gatziu and Dittrich, 1994] event expressions are translated

to Petri nets [Reisig, 1985]. Then, the occurrence of events in the database is represented by placement of tokens in the Petri net. The translation to another formalism is a disadvantage of this approach, because it introduces an extra step in the formalisation. An advantage is the straightforward implementation of a Petri net.

The other approach is a more direct one. The semantics of the event algebra is defined directly in terms of the database history. An example of this is ODE [Gehani *et al.*, 1992]. An advantage is the directness, since the translation is not needed.

The semantics of events in DEGAS is defined in the direct way. As stated at the start of this section, the DEGAS event algebra is a variant of ACP [Baeten and Weijland, 1990], a well-known specification formalism proven in practice [Baeten, 1990]. The ACP concept of matching a process definition and an action trace can easily be translated to the triggering of an event specification by an event trace. The reuse of an existing formalism as an event algebra is a clear advantage of DEGAS. Furthermore, it does not exclude a translation to a different formalism for implementation purposes.

3.4.3 Queries in an Active Database

The link between active and temporal databases, discussed in Section 3.4.2, showed that event specifications can be regarded as conditions on the history of the database. If the history records events, as is the case in DEGAS, then they must also be accessible for queries. Hence, DEGAS queries allow the use of event specifications in the selector.

Event specifications facilitate the formulation of queries like: "Give all credit cards used more than five times last Saturday" or "Give the balance of bank accounts at the time they were debited more than DFL 10,000". Thus, we have an additional means to specify a historical situation in the database, that is independent of time. It is useful to specify that we are interested in a certain situation, without requiring a specific time of occurrence.

A further advantage of the use of event specifications in queries is in the formalisation of the active database. An ECA rule can be considered to be a query-action pair, thus decreasing the number of concepts required. Furthermore, the inclusion of events in a query obviates the need for specific temporal operations in the condition. In fact, the combination event-condition subsumes the temporal conditions in, for example, mono-temporal TSQL [Navathe and Ahmed, 1993].

We know of no earlier work involving events in temporal queries. The work reported in [Claramunt and Thériault, 1995] involves event-oriented queries, but events are a notion from the application, not from the database system itself.

3.4.4 Object Evolution

One of the key features of OODBMSs, as described in Section 3.3, are classes and types. Each object in a database belongs to a class. In most systems, an object's membership of a class is a fixed property. Hence, an object does not migrate from one class to another in the hierarchy. Specialisation of objects, however, is also a dynamic phenomenon. The specialisation is dependent on the *role* of an object. For example, a `person` object is specialised to an `employee` object, because of its role in an `employment` relation. [Gottlob *et al.*, 1996] discuss the extension of object-oriented systems with roles. An extensive conceptual study and formalisation of objects with roles is found in [Wieringa *et al.*, 1995]. There, a distinction is made between static classes, dynamic classes and roles. Objects cannot migrate between static classes, because a static class defines inherent properties of the object. Dynamic classes are dynamic partitions of an object class. Objects can migrate between dynamic classes, possibly subject to constraints. Roles are also dynamic classes but roles do not partition an object class. In addition, an object can play multiple roles at a time.

The relevance of roles for object modelling indicates the need in an object database for a mechanism to dynamically migrate objects from one class to another. Some work has been done in this area. For example, the database programming language Fibonacci [Albano *et al.*, 1993] offers an extensive role mechanism. Roles themselves are part of a hierarchy. Hence, roles can be specialisation of other roles, which gives a relatively complex structure.

An obvious approach is to model roles by inheritance. This is an obvious choice, given that inheritance is the standard specialisation mechanism on most object-based systems. Modelling roles by inheritance, however, has a strong disadvantage, if an object can play multiple roles at a time. 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 [McAllester and Zabih, 1986].

To avoid this combinatorial explosion, each role can be specified separately, while allowing addition of multiple roles at a time. An example is the work on Aspects [Richardson and Schwarz, 1991]. An aspect is a unit of data and behaviour that can be added dynamically to an object. Aspects, however, do not address the link between aspects and relations, as in the `employment` example at the start of this section. Furthermore, although aspects can have other aspects, interaction is not possible between different aspects of the same object. Hence, we cannot model interactions between two roles of the same object. An example would be the use by a `person` of information obtained through his `employee` role in his `investor` role.

In DEGAS, we introduce a simple object extension mechanism, the addon mechanism. This avoids the complications of multiple inheritance. Furthermore, active rules allow object extension to be triggered by events on the database. This

is especially useful to extend an object further, if a combination of addons is present. The addon mechanism is discussed in Chapter 4.

3.5 Conclusion

This chapter presented the areas in database research that are of interest for our research. We started with an overview of active databases, which is the area DEGAS builds on. Object autonomy also gives new perspectives on the connection of active databases to temporal and object-oriented databases, which were also introduced.

Further discussion in this chapter concerns the impact of object autonomy on different issues in active object databases. The maximal encapsulation of object autonomy promotes a consequent application of object-oriented principles to the modularisation of rules in an active databases. Furthermore, the process-oriented view of object autonomy on the formalisation of an active database gives us a model that unifies active and historical databases. A further advantage of this formalisation is that it gives the semantics of rules directly.

The integration of active and temporal database also sheds new light on the specification of temporal queries. Events as temporal conditions allow the specification of historical situations independent of their exact time of occurrence. Another contribution of the DEGAS model is the straightforward object extension mechanism, that allows the implementation of objects with roles.

The next part of this thesis introduces the DEGAS model in full. The presentation of the DEGAS model will make clear, how DEGAS fulfills the contributions described in this chapter.

Part II

Model

Chapter 4

The DEGAS Object Model

Now that the motivation for DEGAS has been discussed in depth, its main concepts can be introduced formally. The basis of DEGAS is the autonomous object. As explained before, these objects can evolve through addons, containers of additional, temporally present, functionality. Furthermore, objects are interrelated through relation objects. For a good understanding of these concepts, we use trading on a stock exchange as an on-going example. This example also serves as an introduction of the DEGAS language.

The introduction of DEGAS concepts is topped off with the DEGAS query language. DEGAS' event specifications offer a new way to formulate temporal conditions. This adds an event specification clause to the usual SQL-like object query format. Object autonomy also leads to the introduction of the quality of a query result. Furthermore, we discuss the object management structures in a DEGAS databases.

4.1 DEGAS Objects

In DEGAS, objects are instances of classes. Hence, a DEGAS object definition specifies an instance of a class. As usual, we distinguish structure and behaviour in a DEGAS object. The structure of an object is determined by the attributes. The behaviour of an object has three components: methods, lifecycles, and rules. Methods specify what an object *can* do. The lifecycles specify what an object *might* do, i.e. what methods it is willing to execute in a certain context, by specifying sequencing of and preconditions on method execution. Rules specify what an object *will* do, by specifying actual actions to be executed in certain situations, defined in terms of events and object states. Thus, methods and lifecycles specify *potential* behaviour of an object, whereas rules describe *actual* behaviour. Traditionally, only potential behaviour is specified in object-oriented databases, often limited to methods only.

The first section of an object specification specifies the attributes of a class. DEGAS supports the types commonly found in object models: simple types, tuple types and power types, i.e., sets. One of types supported is the set of class names in the databases. Hence, other object's attributes can be referred to through path expressions, that are translated to method calls to other objects.

As usual in an object-based model, methods specify the possible state changes of an object. In DEGAS, methods can either change attributes in the object, or call other methods, both local and in other objects. Method calls between objects are by way of non-blocking message passing. This will be further explained in Chapter 6.

An object's lifecycle specifies sequencing of methods and pre-conditions on method invocations. Hence, every method call is checked against the lifecycle of the object. A method call will only be executed, if the current state of the lifecycle allows it. The formalism chosen to specify lifecycles in DEGAS is guarded basic process algebraic expressions [Baeten and Weijland, 1990]. The basic actions in such an expression are method names. Complex expressions are composed using sequential composition, alternative composition, repetition, and parallel merge (or indifference) operators.

Rules in DEGAS follow the usual Event-Condition-Action (ECA) format. Like lifecycles, event specifications in DEGAS are expressed using process algebra. We chose process algebra as an event algebra is, because it is well understood, and has found broad application [Baeten, 1990]. In addition to the operators in a lifecycle, an event specification can use the negation of an event. This denotes any event on the object, except the negated event. As was explained in Section 3.1, the action of an ECA rule is executed on occurrence of the event, if the condition is satisfied. In DEGAS, this check of event and condition is done after every method invocation. The action of a rule is a method call, either local or to a method in another object. Hence, the action is also subject to object lifecycles.

Another way to define a class is generalisation. Generalisation captures commonalities between objects of different classes. There are no instances of a generalisation class, since the class of an object defines its inherent, unchangeable properties. 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.

Objects can be specialised through the *addon* mechanism. An addon defines additional attributes, methods, rules and lifecycles. If an object is extended through an addon, it gains these *transient* capabilities. These cannot be distinguished from the *inherent* capabilities of an object[1]. The capabilities specified

[1]An object with introspection might keep track of its capabilities to determine which capa-

in an addon are lost, when the addon is removed. Since an addon only defines an extension of an object, 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, an object is created, acquires and loses relations, and consequently gains and loses capabilities. An example is an employee that has different capabilities in different jobs.

In DEGAS, relations between objects are objects themselves. Thus, we have a place for data and behaviour of a relation. Furthermore, the fact that a relation object is an object itself, also means that it can engage in relations itself. A more abstract motivation of this objectification is, that a relation is a kind of contract, a view also found in, e.g., NIAM [Nijssen and Halpin, 1990].

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 must be unmarried. Likewise, the termination of a relationship is subject to restrictions. In relations, we often need to store data and behaviour of the relation. An example is the bank account relation between a bank and its clients. This information, and the capabilities to handle termination of the relationship, are stored in a relation object. The capabilities to handle the initiation of a relationship, including the creation of the relation object, can be found in the corresponding relation class object.

An object that engages in a relation is extended using the addon mechanism. Through the addon it acquires the capabilities to handle the relationship. An addon is always added, since an object must have a method to terminate the relation. An example is a person with a bank account. If he is in this relation, he can transfer money to other bank accounts or withdraw money through an ATM[2].

The three meta classes in DEGAS, objects, relation objects, and addons, 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 a DEGAS object. Each class of objects is represented by a class object. These are again typed by the three meta classes. These three layers are depicted in Figure 4.1. They can be characterised as follows:

1. **Instance Level.** This level is the representation of the Universe of Discourse of our information system. Here, objects such as persons, banks and bank accounts can be found.

bilities are permanent and which are transient. A DEGAS object, however, is not equipped with introspection.

[2]Automated Teller Machine

2. **Class Level.** This level contains class objects, that are representations 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.

3. **Meta Class Level.** The meta classes are also represented by objects in the system. This is the highest level in the system. The presence of meta class objects facilitates schema evolution by creating and destroying classes, in analogy to the creation and destruction of objects by class objects.

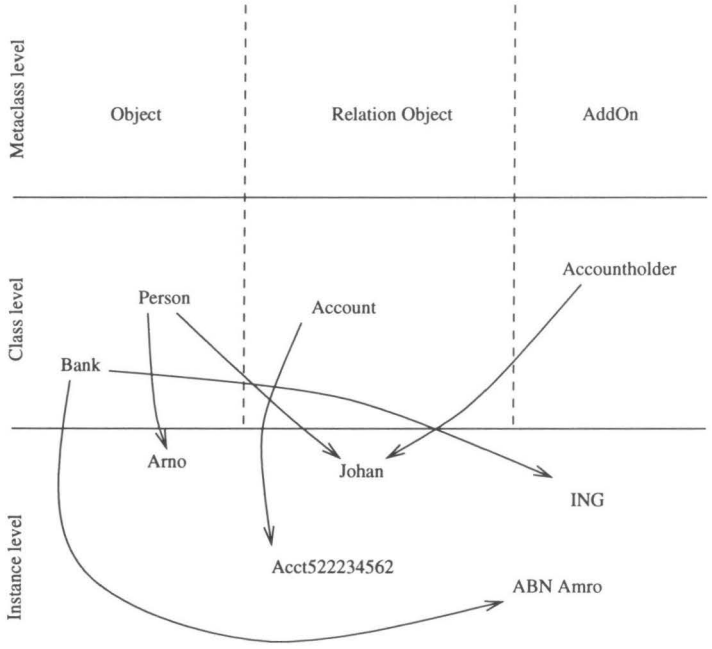


Figure 4.1: Structure of the Object Model

Each class in DEGAS is represented by a class object. Besides recording part of the extent of the class, class objects serve three functions:

1. Creation of new objects
2. Information about the schema of a class
3. Schema updates.

The first two functions are purely local as long as the object schema is fixed. Schema information about objects is necessary to check the correctness of queries. For example, we can test whether an attribute defined in a query actually occurs in that object. Type checking for queries will be discussed in Section 7.4 below.

All three meta classes in DEGAS, i.e., objects, relation objects, and addons, have class objects associated with them. These class objects maintain the extent set of their classes. For example, we can get the set of objects that have an `accountholder` addon through the class object for this addon.

4.2 DEGAS by example

In this section, we introduce the DEGAS concepts by modelling a highly dynamic application, since these are the most challenging to deal with. Trading on the stock exchange[3] is such an application with fast changing data and rapidly evolving relations. New data emerges constantly in the form of buying and selling orders, economic news items through newsreels, et cetera. 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 and sell shares. He can subscribe to a newspaper specialised in news about companies of his interest. Buying and selling of shares goes through a market-maker. If a person wants to buy or sell, he informs the market-maker. Periodically, the market-maker determines the price that balances supply and demand. Buying and selling orders that agree with this price are fulfilled.

We start this example with the market-maker. The market-maker matches supply and demand for his market. Hence, the actions he can execute are to accept buying and selling orders and to try to match these. The data stored by the market-maker is the current price of the share he deals in, which is a real number. This is specified by the DEGAS definition of attributes and methods of an object class `Marketmaker` in Figure 4.2. The methods in this object only contain actions to engage in a relation or actions to extend the object with an addon. An object engages in a relation by sending a method call to the class object of the relation. An example is the `initiateMarketmaker` message sent to the class object of the `Supply` relation. The creation of a DEGAS relation is largely left to the application designer, as is discussed in full detail in Section 7.3.

This defines the basic properties and actions, but we know more about the market-maker. This information is specified in the lifecycle. The lifecycle of a `Marketmaker` object 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. In this process algebraic expression ; denotes sequence, * denotes repetition, and || denotes indifference parallelism.

The specification of the actual execution of actions by a DEGAS object is given by its rules. The behaviour of a market-maker is to register supply and demand,

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

```
Object Marketmaker
Attributes
   currentPrice : real
Methods
   takeSellOrder = {
      SupplyClass.initiateMarketMaker
   }
   takeBuyOrder = {
      DemandClass.initiateMarketMaker
   }
   makeMarket = {
      Extend(SupplyDemand)
   }
Lifecycles
   ((takeSellOrder* || takeBuyOrder*);makeMarket)*
Rules
   On (takeSellOrder||takeBuyOrder) do makeMarket
EndObject
```

Figure 4.2: Specification of the `Marketmaker` object

and clear the market if both are present. A rule, that completes the definition of the `Marketmaker` object, specifies this behaviour.

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 in Figure 4.3

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 kind of 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, specified in the `Relation` clause. These can be used like any other attribute of the relation object. Other information present is the price of the share when it was bought. The definition of the `Share` relation object in Figure 4.4 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, e.g., C++ [Stroustrup, 1991], where the only distinction is between private, public and friend methods.

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

Figure 4.3: Specification of the `Person` object

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

Figure 4.4: Specification of the `Share` relation object

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 in Figure 4.5. An addon temporarily adds capabilities to a DEGAS object. These capabilities are present in the object from the time it is extended by the addon until the addon is removed. The capabilities specified in the addon cannot be distinguished from the inherent capabilities of the object, while they are present. As was discussed in Section 3.4.4, objects have a role in their relations. Since the role is only needed when the object is in the relation, an important use of addons is to model roles in connection with relations.

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

Figure 4.5: Specification of the Shareholder addon

The SupplyClass.initiate action in this addon specification also occurred in the specification of the Marketmaker object. A call to an initiate method is made 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 sends an initiate call to the Supply class object. In response, it sends a takeBuyOrder message to the market-maker to ask, if it is willing to accept the relation. As we can see in the specification of the Marketmaker object, it responds with an initiate call to express its agreement. The Supply class object then proceeds with instantiation of the relation. A further explanation of the way relations are established can be found in Section 7.3.

As we can see 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 influences his decisions as a shareholder. This is achieved by extending the person with a further addon, if he owns shares and reads a newspaper. In Figure 4.6, we give the specification of the `InformedPerson` addon, that extends a `Person` object with a subscription to a newspaper.

```
Addon InformedPerson
Extends Person
Attributes
   subscription : Subscription
   transactionPrice : real
Methods
   goodNews(company : Company) = {
      transactionPrice = subscription.priceAdvice(company)
   }
   badNews(company : Company) = {
      transactionPrice = subscription.priceAdvice(company)
   }
Lifecycles
   ([sender=subscription]goodNews*)
   ([sender=subscription]badNews*)
   (Extend(InformedPerson);Remove(InformedPerson))*
Rules
   On goodNews(company)(t1);goodNews(company)(t2)
      if t2 − t1 ≤ 7 days
      do tryToBuy(company,transactionPrice)
EndAddon
```

Figure 4.6: Specification of the `InformedPerson` addon

The rule definitions in the specifications of `InformedPerson` and `InformedOwner` show 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 specification in Figure 4.7 gives an example of how the informed shareholder deals with bad news. This addon can extend a person, if it has both the `Shareholder` and the `InformedPerson` addons. Hence, the `extends` specification gives two original object names. Please note, that this is not a form of multiple inheritance. It simply specifies, what the addon may assume to be present.

The diagram in Figure 4.8 shows the complete model of the stock exchange example. In this picture, large boxes represent objects and small boxes represent addons. The dashed boxes are relation objects. Please note that the arrows do not imply any arity constraints on the relations. Instead, the arrows simply point to the partners in the relation.

Addon InformedOwner
Extends InformedPerson,Shareholder
Attributes
Key : $\mathcal{P}\langle$ subscription : Subscription, share : Share $\rangle$
Lifecycles
Extend(InformedOwner)*
Remove(InformedOwner)*
Rules
On badNews(company)(t_1);badNews(company)(t_2)
if $(t_2 - t_1) \leq 7$ days **and** transactionPrice(t_2) $\leq$ transactionPrice(t_1)
do tryToSell(transactionPrice)
On goodNews(t_1);badNews(t_2)
if $t_2 - t_1 \leq 7$ days **and** transactionPrice(t_1) = max(transactionPrice, t_1, t_2)
do tryToSell(transactionPrice)
On Remove(ShareHolder) **do** Remove(InformedOwner)
On Remove(Subscription) **do** Remove(InformedOwner)
EndAddon

Figure 4.7: Specification of the `InformedOwner` addon

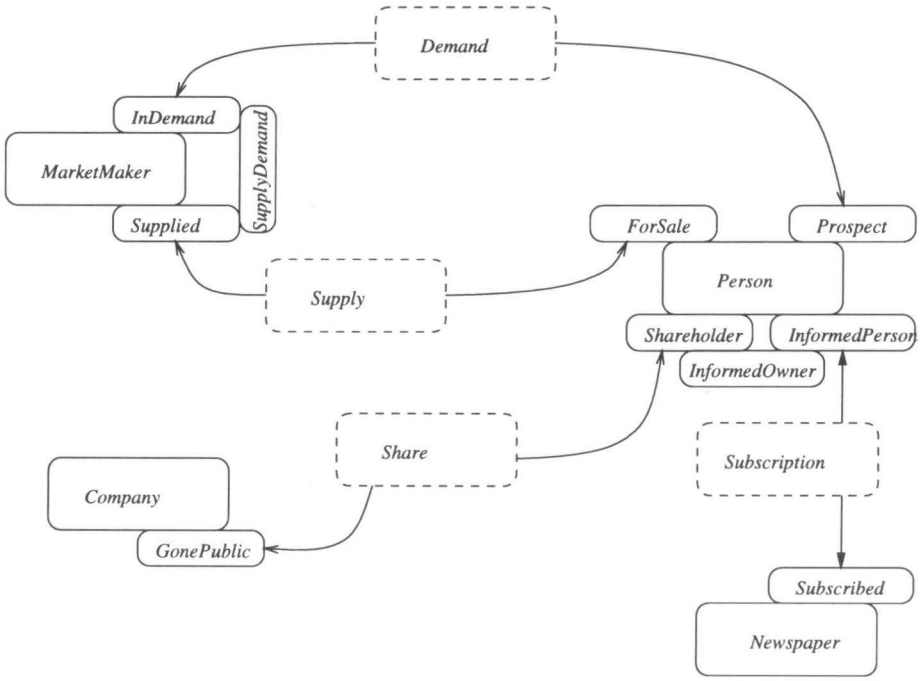


Figure 4.8: The DEGAS model for a financial market

4.3 Syntax of a DEGAS Object

In this section, we give the syntax of the DEGAS data model. Since we showed a number of example DEGAS specifications in the previous section, we only show examples of syntactic constructs that did not occur there. The syntax is given as a BNF grammar. Symbols from the DEGAS language are printed in `teletype` font. Non-terminals are denoted by ⟨NonTerminal⟩ . | denotes a choice. Optional parts are surrounded by rectangular braces: []. Other symbols in the right hand side of a rule are terminal symbols.

Before we proceed with the syntax 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 values for each basic type	⟨BasicValue⟩
A set of Boolean functions on the basic types	⟨BasicCondition⟩
A set of attribute identifiers	⟨AttributeName⟩
A set of parameter identifiers	⟨ParameterId⟩
A set of method identifiers	⟨MethodName⟩
A set of variable names	⟨VariableName⟩
A set of class names	⟨ClassName⟩
An ordered set of label identifiers	⟨LabelName⟩
An linearly ordered set of timestamps	⟨TimeStamp⟩

Please note, that names of attributes and methods must be unique across a complete DEGAS database.

Basic Types and Functions The set of basic types includes the following types. The domains of the types are defined in Section 5.1.

`Oid`	Object identifiers
`Integer`	Natural numbers
`Real`	Real numbers
`String`	Alphanumeric strings
`Boolean`	Truth value
`Time`	Timestamp

Basic functions are defined on the basic types or on a Cartesian product of basic types. The set of basic functions is the following:

+	Addition
−	Subtraction
∗	Multiplication
/	Division

The following Boolean functions are defined on the basic types, where appropriate. For example, on *Oid* only equality and inequality predicates are meaningful.

=	Equality
≠	Inequality
<	less than
≤	less than or equal to
>	greater than
≥	greater than or equal

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

⟨Class⟩ → ⟨ClassHeader⟩ (4.1)
⟨AttributeSection⟩
⟨MethodSection⟩
⟨LifecycleSection⟩
⟨RuleSection⟩
⟨ClassEnd⟩

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⟩ (4.2)
⟨TupleType⟩ → (⟨FieldList⟩) (4.3)
⟨FieldList⟩ → ⟨Field⟩ | ⟨Field⟩ , ⟨FieldList⟩ (4.4)
⟨Field⟩ → ⟨LabelName⟩ : ⟨Type⟩ (4.5)

Class Header The class header indicates the place of the class in the type structure by giving the meta class, i.e., object, relation object, or addon. Further information is the list of subclasses for a generalisation class. 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⟩ (4.6)
⟨ClassHeader⟩ → `Object` ⟨ClassName⟩ `generalises` ⟨ClassList⟩ (4.7)
⟨ClassHeader⟩ → `Object` ⟨ClassName⟩ (4.8)
`Relation` ⟨ClassList⟩
⟨ClassHeader⟩ → `AddOn` ⟨ClassName⟩ (4.9)
`Extends` ⟨ClassList⟩
⟨ClassList⟩ → ⟨ClassName⟩ , ⟨ClassList⟩ | ⟨ClassName⟩ (4.10)

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 specified. There are no syntactical differences between objects, relation objects, and addons in this specification.

Attributes Declaration of attributes is straightforward using the types defined above. Every (relation) object class has an implicit 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` ⟨AttributeList⟩ (4.11)

⟨AttributeList⟩ → ⟨AttributeDecl⟩ (4.12)
| ⟨AttributeDecl⟩ , ⟨AttributeList⟩

⟨AttributeDecl⟩ → ⟨AttributeName⟩ : ⟨Type⟩ (4.13)

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 remove addons from an object. This is explained in more detail in Section 6.2.

⟨MethodSection⟩ → `Methods` ⟨MethodList⟩ (4.14)

⟨MethodList⟩ → ⟨MethodDecl⟩ (4.15)
| ⟨MethodDecl⟩ , ⟨MethodList⟩

⟨MethodDecl⟩ → ⟨MethodName⟩ (⟨ParameterList⟩)= (4.16)
{ ⟨StatementList⟩ }

⟨StatementList⟩ → ⟨Statement⟩ (4.17)
| ⟨Statement⟩ ; ⟨StatementList⟩

⟨Statement⟩ → ⟨AttributeName⟩ := ⟨Expression⟩ (4.18)
| ⟨MethodCall⟩
| ⟨AttributeName⟩ := ⟨MethodCall⟩
| ⟨SetIteration⟩
| `Return` ⟨Expression⟩

⟨Expression⟩ → ⟨AttributeName⟩ | ⟨PathExpression⟩ (4.19)
| ⟨BasicFunction⟩ | ⟨BasicValue⟩

⟨PathExpression⟩ → ⟨PathExpression⟩ . ⟨AttributeName⟩ (4.20)
| ⟨AttributeName⟩

⟨MethodCall⟩ → [⟨PathExpression⟩ .] ⟨MethodName⟩ (⟨ActParamList⟩) (4.21)

⟨SetIteration⟩ → `forall` ⟨VariableName⟩ `in` ⟨AttributeName⟩ (4.22)
`where` ⟨Condition⟩
`do`␣{ ⟨StatementList⟩ }

⟨ActParamList⟩ → ⟨ActParam⟩ | ⟨ActParam⟩ , ⟨ActParamList⟩ (4.23)

⟨ActParam⟩ → [⟨ParameterId⟩ =] ⟨Expression⟩ (4.24)

⟨Condition⟩ → ⟨BasicCondition⟩ (4.25)
| ⟨Condition⟩ `and` ⟨Condition⟩
| ⟨Condition⟩ `or` ⟨Condition⟩

An example of a set iteration is the following method in a `Bank` object. It awards a premium to accounts with a balance higher than a specified limit. The attribute `coffer` represents the bank's own account.

```
Methods
  award(premium : real, premiumLimit : real) = {
    forall acct in Accounts
    where acct.balance > premiumLimit
    do {
      coffer.debit(premium)
      acct.credit(premium)
    }
  }
```

Rules The rules 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.

⟨RuleSection⟩ → `Rules` (4.26)
⟨RuleList⟩

⟨RuleList⟩ → ⟨Rule⟩ | ⟨Rule⟩ , ⟨RuleList⟩ (4.27)

⟨Rule⟩ → `On` ⟨EventSpec⟩ (4.28)
`if` ⟨Condition⟩
`do` ⟨Action⟩

⟨EventSpec⟩ → ⟨Event⟩ [⟨TimeWindow⟩] (4.29)

⟨Event⟩ → ⟨MethodName⟩ [⟨ParameterList⟩] (4.30)
| ⟨TimeStamp⟩
| (⟨Event⟩ + ⟨Event⟩)
| (⟨Event⟩ ; ⟨Event⟩)
| ¬ ⟨Event⟩

		⟨Event⟩ *	
		⟨Event⟩ || ⟨Event⟩	
⟨TimeWindow⟩	→	[⟨TimePairList⟩]	(4.31)
⟨TimePairList⟩	→	⟨TimePair⟩ , ⟨TimePairList⟩ | ⟨TimePair⟩	(4.32)
⟨TimePair⟩	→	(⟨TimeStamp⟩ , ⟨TimeStamp⟩)	(4.33)
⟨Action⟩	→	⟨MethodCall⟩	(4.34)

Lifecycles The lifecycle of an object is defined in the lifecycle section of the class definition. Lifecycles are guarded basic process algebraic expressions. The basic actions are the methods of the object. The guards are conditions on the object state. We can specify a number of expressions in the lifecycle section, each starting on a new line. These expressions are merged into one lifecycle through communication merge, as explained in Section 5.7.2.

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, while an event expression is basically a query on the history of an object.

⟨LifecycleSection⟩	→	`Lifecycles`	(4.35)
		⟨LifecycleList⟩	
⟨LifecycleList⟩	→	⟨Lifecycle⟩ | ⟨Lifecycle⟩ , ⟨LifecycleList⟩	(4.36)
⟨Lifecycle⟩	→	⟨MethodName⟩ [⟨ParameterList⟩]	(4.37)
		| ([⟨Condition⟩] ⟨Lifecycle⟩)	
		| (⟨Lifecycle⟩ + ⟨Lifecycle⟩)	
		| (⟨Lifecycle⟩ ; ⟨Lifecycle⟩)	
		| ⟨Lifecycle⟩ *	
		| (⟨Lifecycle⟩ || ⟨Lifecycle⟩)	

This grammar defines syntactically correct classes. More, however, is needed to get a meaningful hierarchy. For that, we need uniqueness constraints and referential constraints.

Uniqueness Constraints Classes, types, attributes, labels and methods must have unique names.

Referential Constraints The references to other entities in declarations must be correct. More specifically:

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. Typing is discussed further in Section 5.1.

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.

4.4 Querying DEGAS objects

A DEGAS query selects the members of a class that satisfy a specified selection condition or *selector*. A novel feature in the selector is the event specification. This allows the specification of arbitrary temporal conditions. The advantage is in the possibility to specify conditions related independent from specific time points. In a DEGAS query, we can express our interest in an event regardless of the time it occurred.

The temporal dimension of DEGAS is reflected in the result of a query. Since the state of an object includes its complete history, a query against an object state is a query against the history of the object. Hence, it is not sufficient to only give the object identities as a result. The time when the object satisfied the query is also relevant. In DEGAS, a query returns a set of object-history pairs, giving the sub-histories that matched the event specification in the query's selector. The selector consists of two parts, an event expression and a condition. The format of a query is:

Select from ⟨Class⟩
on ⟨EventSpecification⟩
if ⟨Condition⟩
quality ⟨Integer⟩

The full syntax definition of the DEGAS query language is given by the following BNF grammar. Non-terminals in this definition refer to the productions in Section 4.3.

$$\begin{aligned} \langle \text{Query} \rangle \quad &\longrightarrow \quad \texttt{Select\_from}\ \langle \text{ClassName} \rangle \\ &\qquad\ \texttt{on}\ \langle \text{EventSpec} \rangle \\ &\qquad\ \texttt{if}\ \langle \text{CompoundCondition} \rangle \\ &\qquad\ \texttt{quality}\ \langle \text{Percentage} \rangle \end{aligned} \tag{4.38}$$

$$\begin{aligned} \langle \text{CompoundCondition} \rangle \quad &\longrightarrow \quad \langle \text{Condition} \rangle \\ &\qquad\ \mid \texttt{Exists\_in}\ \langle \text{Query} \rangle : \langle \text{Condition} \rangle \end{aligned} \tag{4.39}$$

An example of a DEGAS query is:

Select from ATMcards
on ChangePIN(t1);ChangePIN(t2)
if t2 - t1 ≤ 1 day

This query selects all PIN cards that had their code changed twice within a day. The timestamps associated with the events are bound to the $t1$ and $t2$ parameters. These can be referenced in the condition of the query. The parameters of an event can also be referenced in the query condition, as is done in the following example. This query selects bank accounts involved in a fast transfer of money, hinting at potential illegal activities.

Select from BankAccount
on credit(cr_amount)(t1);debit(db_amount)(t2)
if cr_amount > 10000 **and** db_amount > 10000
and t2 - t1 ≤ 2 days

Since a DEGAS object records its complete history, a query also applies to the complete history. If we wish to restrict it temporally, then we specify a *time window* for the event expression. A time window restricts the part of the history the event expression is checked against. For example, we might be interested only in occurrences of an event during the last fortnight. Here, we are interested in bank accounts that are overdrawn by a single large transaction:

Select from BankAccount
on debit(db_amount)(t)[1 Mar 1997, 15 Mar 1997]
if db_amount > 10000 **and** balance(t) ≤ 0

A further facility in the DEGAS query language is the nested query. A nested query is used to specify conditions over multiple objects. This is done through the `Exists` predicate, that consists of a query and a condition. The condition joins the result of this query to objects in the root class of the query. An example is the following query, that selects the trains with a destination that is the location of a cycle race.

Select from Train
if
Exists in Select from Cyclerace:
Train.destination = Cyclerace.location

Above, we saw one of the novel features of DEGAS queries, viz., events to specify temporal conditions. Another new element is caused by object autonomy. A consequence of object autonomy is that it is decided in an object, whether to answer or not to answer a query. This depends on the object's lifecycle. Furthermore, the inherently distributed nature of DEGAS implies that the underlying network may be partially unreachable. Hence, the answer to a query in DEGAS need not necessarily contain all objects satisfying the query. Therefore, the result of a query is accompanied by an estimate of the quality of a query. By the quality of a query, we mean the number of objects giving a positive answer A relative to the total number of objects in the database satisfying the query selector S:

$$\frac{A}{S}$$

Since the divisor S cannot be given with certainty due to object autonomy, the quality measure returned with a query is an estimate. A user can specify an expected quality for a query through the `quality` clause of a query. For example, in the following query the user is satisfied with an answer including 80 percent of the relevant objects.

Select from ATMcards
on ChangePIN(t1);ChangePIN(t2)
if t2 - t1 $\leq$ 1 day
quality 80 %

In Chapter 6, we discuss the impact of query result quality on query processing in DEGAS.

4.5 Distribution Model

Each DEGAS object has an independent thread of control. In this respect, it follows concurrent object-oriented languages [Agha *et al.*, 1993]. Examples of such languages are Actors [Agha, 1986], POOL-T [America, 1987], and Procol [Bos and Laffra, 1991]. Experience in implementing a database using POOL-T in the PRISMA/DB project [Apers *et al.*, 1992] learns us that a large number of concurrent objects cannot be managed without some structure. Hence, the large number of objects in a DEGAS database is organised in two ways: locality and class. The distribution model describes the assumptions about locality of objects. The organisation by class leads to the three-layered model outlined in Section 4.1. These two structures are used to find the reach the desired objects in DEGAS query processing.

Physically, the objects in a DEGAS database live on networked processing units. This network consists of a number of nodes linked by network connections. In view of the motivation for DEGAS in Chapter 2, we do not make any assumptions about the nature of the computing units and the network connections involved. Both can be anything, from a dedicated parallel server to a mobile phone or from an ATM[4] fibre cable to a GSM[5] wireless link.

Logically, distribution in DEGAS is organised by *sites*. Intuitively, a site is a set of computing units that are close to each other relative to others in the network. The logical network a DEGAS database lives on, is represented by a graph. In this graph, the set of vertices is the set of sites. An edge between two nodes exists, if a connection exists between the two sites.

The topology of a network may change arbitrarily. For example, the network may temporarily become partitioned. For any site, we define the active part of

[4] Asynchronous Transfer Mode
[5] Global System for Mobile communications

the network relative to time. For a network N, time t, and site s, it is denoted by $Active(N, t, s)$

Definition 2 *In a network N, a site s_1 is reachable from s_2 at time t, if there is a path from s_2 to s_1 in $Active(N, t, s_2)$.*

Figure 4.9 gives an example network. In this picture, the full lines indicate the active connections. Consequently, the darker coloured vertices represent the sites reachable from site c.

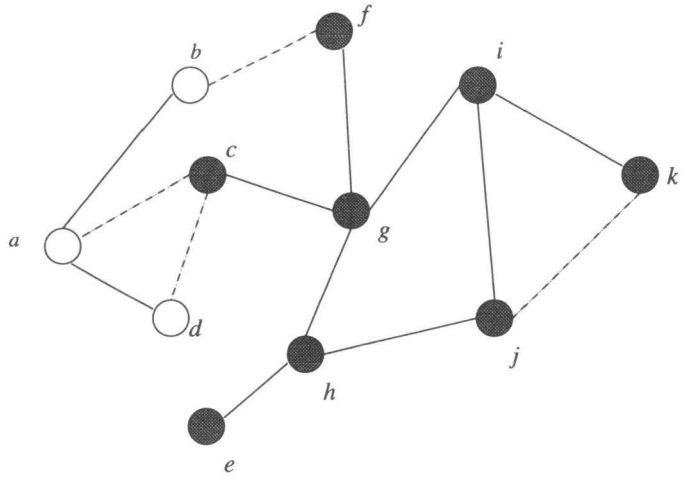


Figure 4.9: Reachability in a network

In DEGAS, each site in the network is represented by a *site object*. A site object functions as a kind of data dictionary. A site object keeps track of the objects residing at its site. Furthermore, it keeps schema information in the form of class objects. To assist query processing, a site object also maintains numerical information about the fragmentation of classes over other sites in the DEGAS database. A further discussion of the workings of site objects is postponed to Section 6.8.

4.6 Conclusion

This chapter introduced the main concepts of the DEGAS model. Object autonomy has two main results. First, every aspect of an object's behaviour is encapsulated. Second, each object has an independent thread of control. The model is based on three meta classes: objects, relation objects, and addons. DEGAS class specifications encapsulate every aspect of an object. In comparison with existent object models, DEGAS extends an object with lifecycles and

rules. The stock exchange example showed how these are used to model the dynamics of information exchange through relations.

The independent thread of control in each object has its impact on query processing. Since it is a local decision to answer a query, the DEGAS query language introduces the quality of a query result. This notion represents the proportion of desired objects in the answer to the query. A further novel facility is the specification of temporal conditions through event expressions. To support query processing a DEGAS database is organised by class and by location in order to get structure in the large number of objects in a database. This is further discussed in Chapter 6.

Chapter 5

Abstract Semantics of DEGAS

The previous chapter gave an introduction to the concepts of the DEGAS model, including a syntax for specifying DEGAS objects. In this chapter, we define the formal semantics of the DEGAS model. In order to give a formal definition of a DEGAS database, a considerable amount of preparatory work is needed.

As a first foundation of the formalisation, we define the type system underlying DEGAS. Based on this type system, we define the effects of DEGAS methods on a tuple of attributes. Then, we discuss the use of time in DEGAS, which results in an initial definition of the history of an object, the pre-history. The pre-history is used to define the interpretation of a DEGAS object and a model of a DEGAS database. The final element defined in advance is a selector, that is used in DEGAS rules and queries.

These preparations allow the formal definition of the dynamic parts of a DEGAS object, viz., methods and rules. In this formalisation, the history plays a central role. For example, whether a method is allowed to execute by a lifecycle is dependent on the history. Lifecycle and rule semantics are defined in process algebraic terms, mapping directly to the event history of the object.

The semantics of method and rule execution lead to a number of constraints on an object history. A database consisting of objects with a valid object history is a valid DEGAS database. This database can be queried using the DEGAS query language. To complete the formalisation of DEGAS, we define the semantics of DEGAS queries.

5.1 Typing

We open the formalisation of DEGAS with the definition of the type system. Typing of attributes and methods in DEGAS is defined following [Balsters and Fokkinga, 1991]. We first give the semantic counterpart of the syntactic construction of the types in Section 4.3.

We start the definition of the DEGAS type system with the basic types.

Definition 3 *A set of basic types B is postulated. B contains the following types:*

Oid	*Object identifiers*
Integer	*Integer numbers*
Real	*Real numbers*
String	*Text strings*
Boolean	*Truth value*

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

Definition 4 *Given a set of basic types B, a hermit type* **1**, *an ordered set of 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* $m \in \mathbf{N}$ *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* $\sigma, \tau \in T$.

DEGAS object specifications, i.e., object, relation object, and addon definitions, define an underlying type, which is a tuple of attributes.

Definition 5 *To each object, relation object, and 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 "*$this : Oid$*". Given a definition D with the following attribute section:*

Attributes
$a_1 : \tau_1$
$a_2 : \tau_2$
...
$a_n : \tau_n$

Then

$$Type(D) = \langle this : oid, a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$$

Additionally, a relation object contains a `Relation` *clause. Given a definition D' with the* `Attribute` *section above and the following* `Relation` *clause:*

Relation $o_1, o_2, \ldots, o_n$

These are also added to the underlying type:

$$Type(D') = \langle this : oid, o_1 : oid, o_2 : oid, \ldots, o_n : oid, a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$$

As an example, we give the underlying type of a `Person` object, defined in Section 4.2, if it is not extended by any addon.

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

A subtyping relationship is defined on the types following [Cardelli, 1984] and [Balsters and Fokkinga, 1991].

Definition 6 *The subtyping relation* $\leq : T \times T$ *is defined as follows:*

1. *if* $\tau \in B$, *then* $\tau \leq \tau$.
2. $Integer < Real$.
3. *Let* $\sigma = (\sigma_1 \rightarrow \sigma_2) \in T$ *and* $\tau = (\tau_1 \rightarrow \tau_2) \in T$. *If* $\tau_1 \leq \sigma_1$ *and* $\sigma_2 \leq \tau_2$, *then* $\sigma \leq \tau$.
4. *if* $\sigma, \tau \in T$ *and* $\sigma \leq \tau$, *then* $\mathcal{P}\sigma \leq \mathcal{P}\tau$.
5. *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* $\sigma_1 \leq \sigma_2$.
6. *if* $\sigma_1 \leq \tau_1$ *and* $\sigma_2 \leq \tau_2$, *then* $\sigma_1 \times \sigma_2 \leq \tau_1 \times \tau_2$.

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

Definition 7 *With each basic type* β *is associated a domain* $D(\beta)$. *The domain of the type* $String$ *is defined by a regular expression.*

$$\begin{aligned} D(Real) &= \mathbb{R} \\ D(Integer) &= \mathbb{Z} \\ D(Boolean) &= \{true, false\} \\ D(String) &= [A\text{-}Za\text{-}z0\text{-}9]+ \\ D(\mathbf{1}) &= \{\varnothing\} \end{aligned}$$

We postulate the existence of a set of object identifiers $D(Oid)$.

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

Definition 8 *The set of basic functions BF consists of the following functions:*

Arithmetic	$+ - * /$
Set operations	$\cup \subseteq \subset \in \setminus$
Equality	$= \neq$
Comparison	$< \leq > \geq$

For each basic function $f \in BF$, we postulate its pre-domain $D_p(f)$.

The domains of the types are defined following [Balsters and Fokkinga, 1991], such that $D(\sigma) \subseteq D(\tau)$ if $\sigma \leq \tau$. First, we define the predomains of the types:

Definition 9 *For each type $\tau \in T$ the predomain of τ, $D_p(\tau)$, is defined as follows:*

1. *The predomain $D_p(\mathbf{1})$ is postulated in Definition 7.*
2. *The predomain $D_p(\beta)$ of a basic type β is postulated in Definition 7.*
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 $f \in BF$ is postulated in Definition 8.*
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 10 *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 β, $D(\beta) = D_p(\beta)$*
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. $D(\sigma \to \tau) = \bigcup_{\sigma' \leq \sigma, \tau' \leq \tau} D_p(\sigma' \to \tau')$

We show that the domains reflects the subtyping relations.

Theorem 1 *Given a two types τ and σ with $\sigma \leq \tau$. Then:*

$$D(\sigma) \subseteq D(\tau)$$

Proof This theorem has been proved for the type system of [Balsters and Fokkinga, 1991], to which the reader is referred. We prove the theorem for the set of basic types B in DEGAS. Additional composite types in the DEGAS type system are the Cartesian product and the set type. We also prove the theorem for these two constructs.

In the set of basic types, we have $Integer < Real$. Since $D(Integer) = \mathbb{Z}$ and $D(Real) = \mathbb{R}$ and $\mathbb{Z} \subseteq \mathbb{R}$, the theorem holds for the basic types.

For the proof of the theorem for set types, we assume that the theorem holds for types other than set types. Given two types $\sigma = \mathcal{P}\sigma'$ and $\tau = \mathcal{P}\tau'$, with $\sigma \leq \tau$. Then, $\sigma' \leq \tau'$. For the domains of σ and τ, we have $D(\sigma) = \mathcal{P}D(\sigma')$ and $D(\tau) = \mathcal{P}D(\tau')$. Since $\sigma' \leq \tau'$, $D(\sigma') \subseteq D(\tau')$, which implies $\mathcal{P}D(\sigma') \subseteq \mathcal{P}D(\tau')$. Hence, $D(\sigma) \subseteq D(\tau)$.

For the proof of the theorem for Cartesian products, we assume that the theorem holds for types other than Cartesian products. Given two types $\sigma = \sigma_1 \times \sigma_2$ and $\tau = \tau_1 \times \tau_2$, with $\sigma \leq \tau$. Then, $\sigma_1 \leq \tau_1$ and $\sigma_2 \leq \tau_2$. For the domains of σ and τ, we have $D(\sigma) = \{(s_1, s_2) | s_1 \in D(\sigma_1) \wedge s_2 \in D(\sigma_2)\}$ and $D(\tau) = \{(t_1, t_2) | t_1 \in D(\tau_1) \wedge t_2 \in D(\tau_2)\}$. Since $\sigma_1 \leq \tau_1$ and $\sigma_2 \leq \tau_2$, we have $D(\sigma_1) \subseteq D(\tau_1)$ and $D(\sigma_2) \subseteq D(\tau_2)$. Consequently, for every element $(s_1, s_2) \in D(\sigma)$, we have $s_1 \in D(\tau_1)$ and $s_2 \in D(\tau_2)$. Hence, every $(s_1, s_2) \in D(\sigma)$ is also an element of $D(\tau)$ and $D(\sigma) \subseteq D(\tau)$. □

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

Definition 11 *Let C be a class and Oid an 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 and $\nexists C$ such that C_1 **generalises** C and C_2 **generalises** C, then $D(C_1) \cap D(C_2) = \emptyset$.*

Methods are typed as well in DEGAS. This is done through function types, like in TM/FM [Balsters *et al.*, 1993]. 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 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 `Accountholder` addon. Before the extension, the underlying type is 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*$\rangle$. This makes typing of methods more complicated than in the standard case. To define the type of a method, we first define how tuple types can be combined. The combination of tuple types is of importance for the definition of object extension through addons. The tuple type composition operator is defined as follows:

Definition 12 *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* $\forall i, 0 < i < n + m : u_i < u_{i+1}$.

Please note that Definition 4 required that labels are unique in DEGAS. Hence, the requirement $\{t_1, \ldots, t_n\} \cap \{s_1, \ldots, s_m\} = \emptyset$ is always satisfied by two tuple types. This unicity requirement can be imposed on DEGAS' type system, because of the absence of multiple inheritance in the language.

The composition of two types can be used in place of the composing types, because the composition is a subtype of each composing type.

Theorem 2 *Given two tuple types T and S, then:*

$$T \otimes S \leq T$$

Proof Recall the definition of the subtyping relation for tuple types:

> 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 $\sigma_1 \leq \sigma_2$.

From Definition 12, we have the following:

$$\begin{aligned} T &= \langle t_1 : \tau_1, \ldots, t_n : \tau_n \rangle \\ S &= \langle s_1 : \sigma_1, \ldots, s_m : \sigma_m \rangle \\ T \otimes S &= \langle u_1 : \nu_1, \ldots, u_{n+m} : \nu_{n+m} \rangle \end{aligned}$$

where $u_i : \nu_i \in \{t_1 : \tau_1, \ldots, t_n : \tau_n, s_1 : \sigma_1, \ldots, s_m : \sigma_m\}$ imposes that all elements of the composition are elements of the composing tuple types. Furthermore, the requirement $\forall i, 0 < i < n + m : u_i < u_{i+1}$ implies that all elements of the composition are unique. Hence:

$$\forall i, 0 < i < n, \exists j, 0 < j < n + m : t_i = u_j \wedge \tau_i = \nu_j$$

This implies the subtyping relation $T \otimes S \leq T$ □

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 the attributes defined in the same addon as the method itself. The typing of methods. however, must take the variability of an object's type into account. 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 [Vreeze, 1991]. 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 13 *Given a method M defined in an addon A that extends a class C with input parameters $in_1 : \tau_1, \ldots, in_n : \tau_n$ and output parameters $out_1 : \sigma_1, \ldots, out_m : \sigma_m$. The type of M is:*

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

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

$$\begin{aligned} \Delta &= Type(Person) \otimes Type(AccountHolder) \\ &= \langle Name : string, Birthday : date, Purse : integer, Account : Oid \rangle \end{aligned}$$

This manner of typing methods preserves subtyping of functions. This is stated in the following theorem:

Theorem 3 *Given two function types, $\tau = \tau_1 \rightarrow \tau_2$ and $\sigma = \sigma_1 \rightarrow \sigma_2$ with $\sigma \leq \tau$. Subtyping is preserved by* DEGAS *method typing:*

$$\begin{array}{l} \forall \rho \in T : \Delta \otimes \rho \times \tau_1 \longrightarrow \Delta \otimes \rho \times \tau_2 \\ \leq \\ \forall \rho \in T : \Delta \otimes \rho \times \sigma_1 \longrightarrow \Delta \otimes \rho \times \sigma_2 \end{array}$$

Proof From the definition of the subtyping relation on Cartesian products in Definition 6, it follows clearly that given types t_1 and t_2:

$$\forall s \in T : t_1 \leq t_2 \Longrightarrow t_1 \times s \leq t_2 \times s$$

Therefore, given

$$\begin{array}{lcl} \sigma & = & \sigma_1 \rightarrow \sigma_2 \\ \tau & = & \tau_1 \rightarrow \tau_2 \\ \sigma & \leq & \tau \end{array}$$

we have:

$$\tau_1 \leq \sigma_1 \wedge \sigma_2 \leq \tau_2$$

and it is straightforward that given a tuple type Δ:

$$\begin{array}{l} \forall \rho \in T : \\ \quad \Delta \otimes \rho \times \tau_1 \leq \Delta \otimes \rho \times \sigma_1 \\ \quad \wedge \\ \quad \Delta \otimes \rho \times \sigma_2 \leq \Delta \otimes \rho \times \tau_2 \end{array}$$

This means that the subtyping relation of functions is preserved by DEGAS method typing. □

5.2 Objects

The previous section defined the type system used for the formalisation of DEGAS. Before we define the effects of method execution in the next section, we define a number of operators for use in the definition of the DEGAS model:

Definition 14 *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.*
4. $Rules(T)$ *yields the set of rules defined in T.*

Similar functions are defined for an object relative to time:

Definition 15 *Given an object O at time t:*

1. *$Attr(O,t)$ yields the set of attributes of O at time t.*
2. *$Meth(O,t)$ yields the set of methods of O at time t.*
3. *$Cycl(O,t)$ yields the set of lifecycles of O at time t.*
4. *$Rules(O,t)$ yields the set of rules of O at time t.*

5.3 Method Semantics

In this section, we define the first element of DEGAS' semantics, viz., the effect of method execution on a tuple of attributes. The results from this section are used to define the semantics of method execution by a DEGAS object in Section 5.7.2.

The effects of method execution are defined in terms of variant interpretations and the semantics of basic functions. Here, we use a simplified notion of an object's interpretation. 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 used in this section, does not include a temporal aspect. Therefore, we refer to it as a *snapshot interpretation.* In Section 5.4, we discuss the full interpretation of a DEGAS object, i.e., relative to time.

A snapshot interpretation of an object maps the attributes of the object to a value in its domain.

Definition 16 *A snapshot 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 α_i such that $k_i \in D(\tau_i)$.

A variant interpretation relates two snapshot interpretations to each other. A variant of a snapshot 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 the attribute α, which is v.

The semantics of method calls is defined in terms of a function M. It yields the interpretation of an object after method execution, given the interpretation before execution and a method call. Hence, its type for an object of class C with type $\tau = Type(C)$ is given by

$$\begin{aligned} M \;:\; & (Oid \longrightarrow D(\tau)) \times Meth(C) \\ & \longrightarrow \\ & (Oid \longrightarrow D(\tau)) \end{aligned}$$

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 17 *Given a statement S and a snapshot interpretation $I(O)$ for object O, $M(S, I(O))$ returns the snapshot 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*

$$M(a_i := BF(p_1, \ldots, p_n), I(O)) = I(O)\{a_i = [\![BF(p_1, \ldots, p_n)]\!]\}$$

   where $[\![\ ]\!]$ denotes the evaluation of the basic function.

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 22.

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

Definition 18 *Let S be a statement, $A : \mathcal{P}\tau$ a set-valued attribute and $I(A) = \{v_1, \ldots, v_n\}$ its interpretation, then*

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

Production 4.22 of the syntax definition defined set iteration:

Forall a **in** A
where C_τ
do S.

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. The semantics of set iteration is defined as follows:

Definition 19 *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_τ is a condition on a variable of type τ 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 the condition C is true for $a : \tau$.

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

Definition 20 *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 21 *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. In this case, we define the return value of the method and the effect on the object executing the method.

Definition 22 *Let $m = S_1; \ldots; S_k;$ **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

$$R(m) = [\![e]\!]$$

where $[\![e]\!]$ denotes the evaluation of e after evaluation of $M(m, I(O))$

5.4 Time in an Autonomous Object

The aspects of the semantics discussed above are all independent of time. Since DEGAS aims at the integration of historical database functionality in an active database, this section proceeds with the temporal semantics of the DEGAS data model. The requirement on the availability of historical data in DEGAS is twofold. First, we need the historic values of attributes to be available. Second, the method calls executed must be available in order to check for triggered rules and to check method calls against lifecycles.

In this section, we discuss the basics of DEGAS' temporal dimension. First, we discuss nature and source of time in a DEGAS database. Then, we give an initial definition of a DEGAS object history. This is an initial definition, because rule execution and lifecycle checking impose a number of constraints on a DEGAS object history to be a valid object history. These constraints are discussed in Section 5.7.

5.4.1 Clocks and Autonomous Objects

Time is relatively simple in DEGAS. As was discussed in Section 3.4.2, 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 any application. Linearity of time means that there is a strict order on all time-stamped events in an object.

Timestamps are always local. In other words, a DEGAS 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 a DEGAS object. First, time increase monotonically. Second, two events never get the same timestamp. These requirements on timestamps are rather loose. It honours one of the two requirements formulated by Lamport in [Lamport, 1978]. 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 is 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, clock synchronisation must take place. The other option is to use one global clock for all objects, which compromises the autonomy of objects. Hence, we do not follow requirement 2 in the DEGAS model. This leads us to the following definition of the clock function in DEGAS:

Definition 23 *The clock of an object O is a function $T_O : \mathbf{N} \rightarrow \mathbf{N}$ that takes as input an event counter and yields as output its current time. T_O is an injective increasing function.*

5.4.2 Pre-history

The history of a DEGAS object records past states of the object. It does this by recording all past states with the method calls that brought the object into the state. Please note, that the history of a DEGAS object is not replayable, since 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.

State Pre-history We now have all definitions to define a state of a DEGAS object at a certain point in time. Following temporal database terminology, this is called the *snapshot state* [McKenzie and Snodgrass, 1991]. A snapshot state records the time the object entered that snapshot state, a valuation for a tuple of attributes, as defined in Definition 16 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. *τ 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 τ in the interval starting at time t.*
4. *MC a 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 a DEGAS object. A state transition can only occur in the history, if it, e.g., respects the lifecycle specified 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 Section 5.7, we define the constraints that 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

$$\forall 0 \leq i \leq n-1 : t_i < t_{i+1} \wedge I(\tau_{i+1}) = M(MC_{i+1}, I(\tau_i))$$

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

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

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 a projection of the state history. The events occurring in a DEGAS object are the method calls 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, such that:*

$$\forall 0 \leq i \leq n, SH(i) = \langle t_i, \tau_i, I(\tau_i), MC_i \rangle : \\ EH(i) \stackrel{def}{=} \langle t_i, MC_i \rangle$$

For use in the definitions on event specification, we define the type of an event history. For an object, the alphabet of the event history is the set of all possible methods occurring in that object.

Definition 27 *Given an object* O *of class* C*. The set of all addons of* C *is denoted by* $AddonSet(C)$*. The set of all potential methods of* O *is defined as follows:*

$$Meth(O) = \{\mu \mid (\mu \in Meth(\gamma)) \wedge (\gamma \in \{C\} \cup AddonSet(C))\}$$

The set of methods at time t *is given, using the set of addons at time* t*, denoted by* $Addons(O,t)$*:*

$$Meth(O,t) = \{\mu \mid (\mu \in Meth(\gamma)) \wedge (\gamma \in \{C\} \cup Addons(O,t))\}$$

Definition 28 *Given an object* O *of class* C *with set of addons* $Addons(C)$ *at time* t*. The type of an event history of* O*, denoted by* $EHist(O)$ *is a string over the alphabet of time-event pairs:*

$$\{\tau \in Timestamp | 0 \leq \tau \leq t\} \times Meth(O)$$

5.5 Interpretation

The next step in the formalisation of DEGAS is the interpretation of a DEGAS object. Using the definition of the interpretation, we can refer to an attribute value of an object at a certain time point. This allows us to check the validity of a condition on an object, which is needed to define the semantics of selectors in Section 5.6.

As stated before, we refer to the state of a DEGAS object at a certain point in time. Hence, an interpretation of an object is defined relative to time:

Definition 29 *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

$$\begin{array}{l} \exists 0 \leq i \leq n, SH(i) = \langle t_i, \tau_i, I(\tau_i), MC_i \rangle : \\ \quad t_i \leq t \\ \wedge \\ \forall i < j \leq n, SH(j) = \langle t_j, \tau_j, I(\tau_j), MC_j \rangle : \\ \quad t < t_j \end{array}$$

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

Definition 30 *$a(t)$ denotes the value of an attribute a in object O at time t. $a(t) = v_a$, if*

$$I(O)(t) \models I(a : \sigma)(t) = v_a$$

where $\models$ denotes the standard logical inference relation [Dalen, 1985].

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

Definition 31 *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 32 *Given a set of classes C, a pre-model for a database at time t $\Gamma(db)(t)$ is a model for db, iff*

$$\begin{array}{l} \forall I(a : \sigma)(t) : \sigma \in C \\ \Longrightarrow \\ \exists o \in db : I(a : \sigma)(t) = o \end{array}$$

From a model of a complete DEGAS database, we get a model for the part of the database related to a specific object. To determine this part, we need the notion of reachability through path expressions.

Definition 33 *Given two objects o_1 and o_2. We can reach o_1 through path expressions from o_2 at time t, denoted by $R(o_2, t)$, if we can construct a path expression $\alpha_1.\alpha_2.\cdots.\alpha_n$, such that:*

$$\begin{array}{l} \alpha_1 \in Attr(o_2, t) \\ \wedge \\ \forall 0 < i < n : \alpha_{i+1} \in Attr(\alpha_1.\cdots.\alpha_i, t) \\ \wedge \\ \alpha_1.\alpha_2.\cdots.\alpha_n = o_1 \end{array}$$

The set of reachable objects is use to define the model

Definition 34 *$\Gamma(O)(t) \subseteq \Gamma(db)(t)$ is the model induced by O. It is defined by*

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

Theorem 4 *Given two objects o_1 and o_2, then:*

$$o_2 \in R(o_1, t) \Rightarrow \Gamma(o_2)(t) \subseteq \Gamma(o_1)(t)$$

Proof For every element $o \in R(o_2, t)$ we can construct a path expression $\alpha_1.\alpha_2.\cdots.\alpha_n$, such that:

$$\begin{array}{l} \alpha_1 \in Attr(o_2, t) \\ \wedge \\ \forall 0 < i < n : \alpha_{i+1} \in Attr(\alpha_1.\cdots.\alpha_i, t) \\ \wedge \\ \alpha_1.\alpha_2.\cdots.\alpha_n = o \end{array}$$

The fact that $o_2 \in R(o_1, t)$ means that we can construct a path expression $\beta_1.\beta_2.\cdots.\beta_m$, such that

$$\begin{array}{l} \beta_1 \in Attr(o_1, t) \\ \wedge \\ \forall 0 < i < m : \beta_{i+1} \in Attr(\beta_1.\cdots.\beta_i, t) \\ \wedge \\ \beta_1.\beta_2.\cdots.\beta_m = o_2 \end{array}$$

As a consequence, if $o \in R(o_2, t)$, the path expression

$$\beta_1.\beta_2.\cdots.\beta_m.\alpha_1.\alpha_2.\cdots.\alpha_n$$

satisfies the requirements of inclusion in $R(o_1, t)$ formulated in Definition 33. Thus, we have:

$$\forall o \in R(o_2, t) : o_2 \in R(o_1, t) \Longrightarrow o \in R(o_1, t)$$

Hence:

$$R(o_2, t) \subseteq R(o_1, t)$$

which means that

$$\begin{array}{l} \bigcup_{o \in R(o_2,t)} I(o)(t) = \Gamma(o_2)(t) \\ \subseteq \\ \bigcup_{o \in R(o_1,t)} I(o)(t) = \Gamma(o_1)(t) \end{array}$$

□

The model of an object over an interval is obtained by taking the models at the time points in the interval together through a direct sum, denoted by $\oplus$.

Definition 35 *Given an object O and an interval (t_{start}, t_{end}). We define a model $\Gamma(O, (t_{start}, t_{end}))$ for O during (t_{start}, t_{end}), as follows:*

$$\Gamma(O, (t_{start}, t_{end})) = \bigoplus_{t \in (t_{start}, t_{end})} \Gamma(O)(t)$$

5.6 Selectors

Selectors play an important role in two key elements of DEGAS, rules and queries. As defined in Chapter 4, a selector is an event-condition pair. An object is selected by a selector, if the event occurs in the history and the condition is satisfied. Since this situation can occur multiple times in the history of a DEGAS object, a selector returns the set of sub-histories that satisfy the selector.

Clearly, the definition of selectors is based on the event history of an object, defined in Section 5.4, and the interpretation of an object, defined in Section 5.5. Selectors are used to define the semantics of DEGAS rules in Section 5.7.3 and to define the semantics of DEGAS queries in Section 5.9.

5.6.1 Event Specification

An event specification consists of an event expression and a time window. An event expression is a process algebraic expression [Baeten and Weijland, 1990] over an alphabet, that is the union of the set of method names of the object and timestamps.

Definition 36 *An event expression is a process algebraic expression over a set of basic actions:*

$$\mathcal{E} = Meth(O)$$

The type of an event expression is denoted by $EventExpr$. *An event expression matches an event history, if a sub-history exists that is a trace of the process specified by the event expression, as defined in [Baeten and Weijland, 1990, Chapter 7].*

The process algebraic operators used, and their meaning are given in the following table. Please note that, like in ordinary calculus, $\sum$ represents a sequence of +'s.

Sequence	$A;B$	A followed by B
Choice	$A+B$	A or B
Repetition	A^*	One or more times A
Merge	$A\|\|B = A;B + B;A$	A and B in parallel
Negation	$\neg A = \sum_{e \in \mathcal{E} \setminus \{A\}} e$	An event that is not A

An additional operator used in an event expression, that is not an action, is the $\perp$ symbol. It denotes the end of the event history, i.e., an event expression ended with $\perp$ only matches the tail of an event history.

Since multiple matching sub-histories may be found, the result of testing an event expression on an event history is a set of matching sub-histories. As an example, consider the event history:

$$\langle 1,e\rangle;\langle 3,f\rangle;\langle 8,g\rangle;\langle 10,a\rangle;\langle 11,a\rangle;\langle 12,a\rangle;\langle 13,b\rangle;\langle 14,c\rangle;$$
$$\langle 22,d\rangle;\langle 33,e\rangle;\langle 34,f\rangle;\langle 39,b\rangle;\langle 40,c\rangle$$

Testing the event expression $b;c$ on this event history yields the set

$$\{\langle 13,b\rangle;\langle 14,c\rangle,\ \langle 39,b\rangle;\langle 40,c\rangle\}$$

The event expression $b;c;\perp$ is only matched by the tail of the event history. Its result is:

$$\{\langle 39,b\rangle;\langle 40,c\rangle\}$$

Another example is the event expression $a^*;b$ that is matched by the set

$$\{\langle 10,a\rangle;\langle 11,a\rangle;\langle 12,a\rangle;\langle 13,b\rangle,\ \langle 11,a\rangle;\langle 12,a\rangle;\langle 13,b\rangle,\ \langle 12,a\rangle;\langle 13,b\rangle\}$$

An event expression is extended to an *event specification* by adding a time window to it. A time window specifies a subset of the history by giving pairs of timestamps that bound the intervals comprising the sub-history. Informally speaking, its effect is that the event expression is only checked against this part of the history.

Definition 37 *The type of an interval specification is denoted by* $TimeSpec$, *defined as:*

$$TimeSpec = \mathcal{P}(Timestamp \times Timestamp)$$

where:

$$\forall T : TimeSpec : (t_1, t_2) \in T \Longrightarrow t_1 \leq t_2$$

A time window is a function that yields a set of sub-histories, given an event history and a time specification.

Definition 38 *Given an object* O *of class* C, *then the time window function* TW *on* O *is typed as follows:*

$$TW : TimeSpec \times EHist(C) \longrightarrow \mathcal{P}EHist(C)$$

The result of a time window function's application is defined as follows:

$$\begin{array}{l}\forall \epsilon : EHist(C), TS : TimeSpec, EH : EHist(C), n \in \mathbf{N} : \\ \quad \epsilon = \epsilon_1;\ldots;\epsilon_n \\ \quad \sigma \in TW(TS, EH) \\ \quad \Longleftrightarrow \\ \quad \exists (t_1, t_2) \in TS, \forall 1 \leq i \leq n : \\ \qquad t_1 \leq \epsilon_i.t \leq t_2\end{array}$$

As an example, suppose that we again have the following event history:

$$\begin{array}{l}\langle 1,e\rangle;\langle 3,f\rangle;\langle 8,g\rangle;\langle 10,a\rangle;\langle 11,a\rangle;\langle 12,a\rangle;\langle 13,b\rangle;\langle 14,c\rangle; \\ \quad \langle 22,d\rangle;\langle 33,e\rangle;\langle 34,f\rangle;\langle 39,b\rangle;\langle 40,c\rangle\end{array}$$

The time specification $\{(1,9)\}$ gives us the following sub-history as a result:

$$\langle 1,e\rangle;\langle 3,f\rangle;\langle 8,g\rangle,\langle 9\rangle$$

Multiple elements in the time specification means that the result has multiple elements. For example, the time specification $\{(1,9),(20,35)\}$ gives us the following result:

$$\langle 1,e\rangle;\langle 3,f\rangle;\langle 8,g\rangle,\langle 9\rangle$$
$$\langle 20\rangle;\langle 22,d\rangle;\langle 33,e\rangle;\langle 34,f\rangle;\langle 35\rangle$$

Given these preliminary definitions, we can now turn to the definition of a complete event specification. An event specification has two components, an event expression $Expr$ and an interval specification TS:

$$Expr[TS]$$

An event specification is triggered, if one of the sub-histories in the time window parses the event expression correctly. The reader is referred to [Baeten and Weijland, 1990, Chapter 7] for an explanation of matching process specifications with process traces, i.e., matching event expressions with event histories.

Definition 39 *Given an object O. An event specification $E = Expr[TS]$ is triggered, if $Expr$ matches (as defined in Definition 36) one of the sub-histories in $TW(TS, EHist(O))$, where $EHist(O)$ is the event history of O. The function $EventTest(E,O)$ returns the set of matching sub-histories of E. The type of this function is:*

$$EventTest : (EventExpr \times TimeSpec) \times C \rightarrow \mathcal{P}EHist(C)$$

For example, consider the event history given above and the following event specification:

$$f;g[(1,9),(20,35)]$$

It would be triggered in the first sub-history. The function $EventTest(E,O)$ returns the singleton set

$$\{\langle 3,f\rangle;\langle 8,g\rangle\}$$

An event specification can be triggered by multiple occurrences of an event. In this case, we will get multiple matching sub-histories. For example, the event specification

$$e;f[(1,10),(20,35)]$$

yields as the result of $EventTest(E,O)$ the following set:

$$\{\langle 1,e\rangle;\langle 3,f\rangle,\ \langle 33,e\rangle;\langle 34,f\rangle\}$$

5.6.2 Condition

The other part of a selector is the condition. A condition is a predicate on the model induced by the object. In other words, it can contain local attributes and attributes of other objects, provided these are reachable through path expressions. Hence, the typing is as follows:

Definition 40 *A condition C defined in an addon A extending class C is a boolean function*

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

Satisfaction of a condition by an object relative to time is then defined as follows:

Definition 41 *Given a condition C on an object O and a sub-history EH of O. The timestamp of the first event in EH is denoted by $t_{EH,start}$. The timestamp of the last event in EH is denoted by $t_{EH,end}$. C is satisfied by O during EH, denoted by $C(O, EH)$, iff*

$$\Gamma(O, (t_{EH,start}, t_{EH,end})) \vDash C$$

where $\vDash$ denotes the standard logical inference relation [Dalen, 1985].

5.6.3 Selection

An object is selected by a selector, if the event specified in the **On** clause occurs and the condition in the **if** clause is satisfied during the matching sub-history of the event.

Definition 42 (Selection of an object) *An object O of class C is selected by a selector $S = \langle E, C \rangle$ with an event specification E and a condition C, iff*

$$\exists EH \in EventTest(E, O) : C(O, EH)$$

The selection set $Selected(S, O)$ is defined accordingly:

$$Selected(S, O) = \{EH \in EventTest(E, O) \mid C(O, EH)\}$$

5.7 From pre-history to history

The work in the previous sections allows us to formalise the dynamic aspects of a DEGAS object, viz., type evolution, method execution and rule execution. In this section, we define their semantics building on the elements defined earlier in this chapter. Furthermore, these semantics yield a number of constraints on the history of a DEGAS object. These constraints lead to the definition of a valid object history and a valid DEGAS database in Section 5.8.

5.7.1 Type Evolution

The type of a DEGAS object can change over time through the addon mechanism. Hence, 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:*

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

where $Extend(\alpha)$ extends O with an addon α.

The effect of $Extend(\alpha)$ on the set of addons is as follows:

Definition 43 *Given an object O of class C and an addon $\alpha \in AddonSet(C)$. The effect of an action $Extend(\alpha)$ on the set of current addons $Addons$ is given by:*

$$Addons(t_{i+1}, O) = Addons(t_i, O) \cup \{\alpha\}$$

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:*

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

where $Remove(\alpha)$ is the action to remove addon α.

Definition 44 *Given an object O of class C and an addon $\alpha \in AddonSet(C)$. The effect of an action $Remove(\alpha)$ on the set of current addons $Addons$ is given by:*

$$Addons(t_{i+1}, O) = Addons(t_i, O) \setminus \{\alpha\}$$

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

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

5.7.2 Method Execution and Lifecycles

Execution of a method is the only way to bring a DEGAS object from one state to another. Therefore, every state of the object must be the result of the application of a method to the previous state. In addition, execution of methods and rules must conform to the lifecycles. As we saw above, lifecycles are event expressions, where an event can be guarded by a condition.

Definition 45 *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.

$$\mathcal{M} = Meth(O)$$

and the guard conditions are of the type defined in Definition 40.

The semantics of lifecycles is formulated in process algebraic terms [Baeten and Weijland, 1990].

Definition 46 *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 γ *defined by:* $\forall \alpha \in \mathcal{M} : \gamma(\alpha, \alpha) = \alpha$, *where* $\mathcal{M}$ *is the event alphabet of* C *as defined in Definition 45.*

In process algebra, a communication function γ specifies synchronisation between two processes. $\gamma(A, B) = C$ means that the actions A and B have to take place simultaneously and are 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 define the set of lifecycles of an object relative to time.

Definition 47 *Given an object O of class C. The set of lifecycles at time t is defined as follows:*

$$\mathcal{C}ycl(O,t) = \mathcal{C}ycl(C) \cup \bigcup_{\alpha \in Addons(O,t)} \mathcal{C}ycl(\alpha)$$

The set of lifecycles of an object is translated to a single process specification. Method execution will be checked against this process.

Definition 48 *Given $\mathcal{C}ycl(O,t)$, the set of lifecycles of an object O at time t. The lifecycle of O at time t is given by:*

$$LC_O(t) = |_{C \in \mathcal{C}ycl(O,t)} C$$

with communication function γ defined by: $\forall \alpha \in \mathcal{M} : \gamma(\alpha, \alpha) = \alpha$, where $\mathcal{M}$ is the event alphabet of C as defined in Definition 45.

Different alternatives exist for the composition of the complete lifecycle of a DEGAS object from the set of specified lifecycles. These alternatives are 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 μ occurs in lifecycles C_1 and C_2. If a call to μ 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 γ is defined by $\forall \alpha \in \mathcal{M} : \gamma(\alpha, \alpha) = \alpha$, where $\mathcal{M}$ is the event 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 has obvious advantages 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 are not allowed to 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 only extend the original lifecycle, e.g., by interspersing 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 [Akker and Siebes, 1995b]. 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:

$$A;C;B$$

Now, we have a problem 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 Communication merge as a composition operator for lifecycles gives us inherent redefinition, which is needed by addons. 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 within an object or addon specification 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.

Lifecycle Checking A method is executed, if it does not violate the lifecycles imposed on the object. Hence, 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 by the lifecycle. If this is true, the method call is executed and appended to the history. If the method call does not satisfy the lifecycle of an object, it is removed from the method queue and discarded.

As a result of addon extension and deletion, the set of events, i.e., methods, is not fixed over the lifetime of an object. This means that we might encounter events in the history, that are currently not defined on the object, since they are part of a removed addon, and consequently are not present in the current lifecycle. To cater for the deletion of addons, the lifecycle check abstracts from the events not currently present in the object. In other words, if an event is a method of a removed addon, then it is disregarded for the check of a method call against the current lifecycle. Abstraction is defined using the ACP abstraction operator ∂.

Definition 49 *A method call $m(p_1,\ldots,p_k)$ is executed on an object O with state history $SH = SH(0);\ldots;SH(n)$ at time t, iff $\partial_H(EH;m(p_1,\ldots,p_k))$ is a prefix*

of a trace that matches the process $LC_O(t)$ and $C(t,O)$, where $H = Meth(O,t)$. The resulting new state of the object is

$$SH' = SH; \langle t, \tau_n, M(m(p_1,\ldots,p_k), I(\tau_n)), m(p_1,\ldots,p_k)\rangle$$

This requirement on method execution is translated to the following constraint on the state history of a DEGAS object.

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

5.7.3 Rule Execution

As explained before, rules in DEGAS use the event-condition-action format. As we described in Section 5.6, an event-condition pair is a selector. Hence, for the definition of its semantics, a rule is considered to be a selector-action pair.

Definition 50 *A rule R is a triple $\langle S, A\rangle$, where $S = \langle E, C\rangle$ is a selector with an event expression E and a condition C. A is a method call, as defined in Production 4.21.*

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

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

A rule is triggered, if the object is selected by the selector of the rule. Please note, that encapsulation of rules means that the selector is applied to a single object in this case.

Definition 51 *A rule $R = \langle S, A\rangle$ with a selector S and an action A is triggered on an object O, if $Selected(S,O) \neq \emptyset$.*

From this definition follows the definition of the set of triggered rules.

Definition 52 *The set of triggered rules $\mathcal{R}$ at time t on object o is:*

$$\mathcal{R} = \{\rho \in Rules(o,t) \mid \rho = \langle S, A\rangle \wedge Selected(S,O) \neq \emptyset\}$$

where $Rules(o,t)$ denotes the set of rules in object o at time t.

At first sight, one would expect rule processing to result in a constraint that one of the rules triggered by a method must be executed. The fact, however, that actions of rules must also obey an object's lifecycle means that a rule's action need not necessarily be executed.

5.8 A DEGAS Database

The preceding sections in this chapter defined everything needed to formalise a DEGAS database. The typing system served as a foundation to the definition of method execution. Both were used in the preliminary definition of a DEGAS object history. In order to define the constraints on a DEGAS object history, we formalised the interpretation of a DEGAS object, which allowed us to define selectors in DEGAS. After this, we were able to define the semantics of method and rule execution. These resulted in a number of constraints on the history of a DEGAS object.

A valid object history of a DEGAS object is defined as follows:

Definition 53 *A state pre-history SH of an object O is a state history for O, iff it satisfies all constraints defined in this Chapter.*

A DEGAS database is a collection of objects with a valid object history and correct references between objects.

Definition 54 *A collection of objects Ω is valid* DEGAS *database at time t, if:*

1. *Each $o \in \Omega$ has a valid object history.*
2. *For each time point $o \leq \tau \leq t$, we have a valid model $\Gamma(\Omega)(\tau)$.*

5.9 Queries

A DEGAS query collects a set of objects in the root class of a query, specified in the `select from` part, that satisfies the selector. Object autonomy implies that some objects will be in a state where they are not willing to respond to a query. Furthermore, as a result of the volatility of the network, a query might not reach all objects. The semantics of the DEGAS query language includes these issues. The abstract semantics of the DEGAS query language is set-based. The main reason to use this kind of semantics is the need to include reachability and willingness to respond of objects in the DEGAS query semantics. These notions cannot be included in a semantics based on an object algebra [Alhajj and Arkun, 1993, Özsu and Straube, 1991, Shaw and Zdonik, 1990].

A query consists of a class and a selector:

Definition 55 *A query Q is a pair $\langle C, S \rangle$, where C is a class and S is a selector on that class.*

Recall from Section 4.4, that the syntactic equivalent of a query $Q = \langle Class, S \rangle$, with $S = \langle E, C \rangle$ is:

Select from *Class*
on *E*
if *C*

The selector of a query has one extension relative to the selector defined in Section 5.6. The `exists` clause makes it possible to connect different classes through a nested query. The result of this query is related to objects in the root class of the query. The semantics of a nested query is defined as follows:

Definition 56 *Given a query Q_1 with a nested query:*

Select from C_1
on E_1
if C_1' ***and Exists in*** $Q_2 : C(O_1, O_2)$

where $Q_2 = \langle C_2, S_2 \rangle$. We define two selection sets:

$$S_1 = \{o \in C_1 \mid Selected(\langle E_1, C_1' \rangle, o) \neq \emptyset\}$$
$$S_2 = \{o \in C_2 \mid Selected(S_2, o) \neq \emptyset\}$$

Let EH_1 be a matching sub-history of $O_1 \in S_1$ and EH_2 a matching sub-history of $O_2 \in S_2$. The timestamps $t_{EH_1,start}$, $t_{EH_1,end}$, $t_{EH_2,start}$, and $t_{EH_2,end}$ are defined the same as in Definition 41. This predicate is satisfied by O_1 and O_2, iff

$$\Gamma(O_1, (t_{EH_1,start}, t_{EH_1,end})) \oplus \Gamma(O_2, (t_{EH_2,start}, t_{EH_2,end})) \vDash C(O_1, O_2)$$

where $\vDash$ denotes the standard logical inference relation [Dalen, 1985] and where $\oplus$ denotes a direct sum.

Due to network failures, it is possible that a query does not reach all objects. Hence, the set of objects the query is applied to is restricted to the set of objects reachable from the site where the query is issued.

Definition 57 (Reachability of an object) *An object o at site s_1 is reachable at time t for a query Q issued at site s_2, denoted by $Reachable(Q, o, t)$, if s_1 is reachable from s_2 as defined in Definition 2.*

To answer a query an object must be in a state where it is willing to answer the query. This means that its lifecycle must allow the execution of the `CheckSelector` method at the time it takes the method call from its message queue.

Definition 58 (Willingness to respond) *Given an object o and a query Q that sends a `CheckSelector` method call q to o. o is willing to respond to Q at time t, denoted by $Willing(Q, o, t)$, if the method call q satisfies the lifecycle of O at time t, where t is the time o takes the call to q from its message queue.*

Using the notions of reachability and willingness, we define the result of a query. The result is a set containing object - history pairs, where an element $(O, Hist)$ means that the query is satisfied by sub-history $Hist$ of object O. O must be in the set of willing and reachable objects.

Definition 59 *A query $Q = \langle S, C \rangle$ issued at time t_q. It is a function*

$$ClassName \times (EventExpr \times TimeSpec \times Condition) \\ \longrightarrow \\ \mathcal{P}(Oid, EHist(C))$$

The resulting set of objects is the set

$$Result(Q) = \{(O, EH) \mid O \in C \wedge EH \in Selected(S, O) \\ \wedge Reachable(Q, O, t_q) \wedge Willing(Q, O, t_q)$$

5.10 Conclusion

In this chapter, we discussed the abstract semantics of DEGAS. The fundamentals are the typing of objects, the semantics of method execution, and the nature of time in DEGAS. These are used to define the central notion of the DEGAS semantics, the object history. An object history is a sequence of the past object states. Since the history is the result of execution of method calls and rules, the semantics of these dynamic aspects determine constraints on the history.

The use of process algebra for lifecycle and rule specification gives a straightforward way to define their semantics. The event history must be a trace of the process specified by the lifecycle. Reversely, a rule is triggered if a sub-history of the event history is a trace for the event specification in the rule's selector.

The formal definition of DEGAS queries is set-based. A novel feature is the use of event specification as a temporal condition. The query semantics includes the notions of reachability and willingness, that are particular to the DEGAS model. These lead to a notion of query result quality, which is further examined along other operational aspects of DEGAS in the next chapter.

Chapter 6

Functional Specification of DEGAS

Chapter 5 gave the formal definition of the DEGAS model. In this chapter, we give a functional specification of the elements needed to implement a DEGAS database. A functioning DEGAS database requires a number of elements to be present. Except for a layer providing basic object functionality, these are all DEGAS objects. Hence, the specification of the basic DEGAS object is the central element of this chapter.

A specialisation of a DEGAS object is the relation object. Its function implies a number of additional requirements on the data it stores and the actions it implements. Besides instances of objects and relation objects, a DEGAS database needs class objects to manage objects, relation objects, and addons. Furthermore, the distribution model is implemented by site objects. For each element of a DEGAS database, we give the data structures required, the primitive actions offered, and their execution.

Further specification given in this chapter is a functional specification of query processing. Here, we give the objects required to implement DEGAS query processing following the same approach as for the other aspects of a DEGAS database.

6.1 Preliminaries

Before we give the formal specification of a DEGAS database, we define shorthands for a number of often used types. Furthermore, we define the notation for a list. Please note, that the types in this chapter are a form of pseudo-typing, since the specified implementation is outside the scope of the DEGAS type system.

A type used in this chapter is the type *MethodCall*. It consists of a method name and a list of parameters:

$$MethodCall = string \times [\langle parameterName, parameterValue \rangle]$$

The type of *parameterValue* depends on the type of the method parameter.

A further type is the type of a selector, denoted by *SelectorType*. Recall from Section 5.6, that it is defined as:

$$SelectorType = (EventExpr \times TimeSpec) \times Condition$$

A final matter of notation serves to specify the presence of a set of named capabilities in an object, where the name is dependent on the contents of the set. An example is the `get` method associated with an attribute. In this case, the name of the attribute is part of the name of the method, because the set of attributes is fixed for a given object specification. The notation for the presence of named capabilities, where a part of the name `fixedpart` is fixed and another part is a variable `Name` over a set *Set*, is as follows:

⋆ ⋆ ⋆ For each `Name` $\in$ *Set* : ⋆ ⋆ ⋆
fixedpart<Name>

For example, a DEGAS object has a `get` method for every attribute. This is written as follows:

⋆ ⋆ ⋆ For each `Attribute` $\in$ *Attr* : ⋆ ⋆ ⋆
get<Attribute>

6.2 Objects

The basic building block of a DEGAS implementation is the object. In this section, we specify the working of the basic DEGAS object. First, we specify the information recorded in a DEGAS object. Then, we give the primitive actions of a DEGAS object. The main result of this section is the specification of the execution cycle of an object. This cycle implements all dynamic aspects of a DEGAS object, viz., method execution, rule execution, and query processing.

6.2.1 Data Structures

A basic DEGAS object contains the following data structures:

State History The object records its complete historical state, as was defined in Definitions 25 and 53. It is denoted by *SH*.

Query Queue The query queue is the queue for incoming query messages. In the specification of a DEGAS object it is denoted by $\mathcal{Q}$. The incoming queries are calls to the `CheckSelector` method. Besides the requested selector, we record in $\mathcal{Q}$ the identity of the sender and of the reply's recipient. Furthermore, a query has an identity qid to allow the `Site` object to process multiple queries at a time. This is explained further in Section 6.8. Consequently, the query queue is specified as follows:

$$\mathcal{Q} \ : \ [\langle sender : oid, replyTo : oid, qid : integer, selector : SelectorType \rangle]$$

External Method Queue Method calls from other objects are queued in the external method queue, denoted by $\mathcal{M}_{ext}$. Each entry records the sender of the call, the reply's recipient, the name of the method called, and the list of parameters. The identity of the sender is used in the lifecycle, while the answer of a method is sent to the reply's recipient. This differentiation is used to process queries in the `CheckSelector` action and to efficiently evaluate path expressions using the `Get` action. Consequently, $\mathcal{M}_{ext}$ is specified as follows:

$$\mathcal{M}_{ext} \ : \ [\langle sender : oid, replyTo : oid, MC : MethodCall \rangle]$$

Internal Method Queue Method calls from other methods in the object are queued in the internal method queue. It is denoted by $\mathcal{M}_{int}$. Otherwise, the internal method queue is identical to the external method queue.

$$\mathcal{M}_{int} \ : \ [\langle sender : oid, replyTo : oid, MC : MethodCall \rangle]$$

Reply Box The answer to the evaluation of a path expression is put into the reply box. It is denoted by $\mathcal{RB}$. It can contain a single value of any type.

$$\mathcal{RB} \ : \ Value$$

Capability Sets Each category of capabilities is represented in the DEGAS object by a set. To illustrate the discussion, we use the specification of a `Person` object from Section 4.2. It is repeated in Figure 6.1.

Attribute Set This set contains the current attributes of the object. It is denoted by $Attr$. It is the materialisation of the function $Attr(O,t)$ defined in Definition 15. In the unextended `Person` object in Figure 6.1, it is:

$$Attr \ = \ \{\texttt{name, birthday, birthplace}\}$$

Method Set This set contains the current methods of the object. It is denoted by $Meth$. It is the materialisation of the function $Meth(O,t)$ defined in Defini-

```
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 = {
      Extend(InformedOwner)
   }
Lifecycles
   (tryToBuy)*
   ((extend(Shareholder)||extend(InformedPerson));useNews)*
Rules
   On (extend(Shareholder)||extend(InformedPerson))
      do useNews
EndObject
```

Figure 6.1: Specification of the `Person` object, given earlier in Figure 4.3.

tion 15. In the unextended Person object in Figure 6.1, it is:

$$Meth = \{\texttt{tryToBuy, readPaper, useNews}\}$$

Lifecycle Set This set contains the current lifecycles of the object. It is denoted by $Cycl$. It is the materialisation of the function $Cycl(O,t)$ defined in Definition 15. In the unextended Person object in Figure 6.1, it is:

$Cycl = \{$
(tryToBuy)*,
((extend(Shareholder)||extend(InformedPerson));useNews)*
}

Rule Set This set contains the current rules of the object. It is denoted by $Rules$. It is the materialisation of the function $Rules(O,t)$ defined in Definition 15. In the unextended Person object in Figure 6.1, it is:

$Rules = \{$
On (extend(Shareholder)||extend(InformedPerson))
do useNews
}

This This attribute contains the object's own identity.

$$This : Oid$$

It cannot be changed, but can be referenced as a normal attribute.

6.2.2 Primitive Actions

The following actions are implemented by a DEGAS object:

CheckSelector This action checks the satisfaction of an event - condition pair. It implements the $EventTest$ function defined in Definition 39. The behaviour of CheckSelector is dependent on the sender of the action. If the CheckSelector action is invoked from outside for query processing, it must check whether the sender of the message is allowed access to the attributes in the condition.

The input of CheckSelector is a selector, a sender and a replyTo. Its output is a set of matching sub-histories. It is specified as follows:

$$SelectorType \times Oid \times Oid \longrightarrow \mathcal{P}EHist$$

If `CheckSelector` is called from outside, i.e., `Sender` is not the object itself, it executes `get` actions to check access of `Sender` to the attributes in the condition. Then, the tests in `CheckSelector` are executed in the sequence discussed in Section 5.6. First, the event specification is matched against the event history. Then, the condition is tested on each matching sub-history. A call to `CheckSelector` is not recorded in the history, because it is not a method.

The algorithm executed by the `CheckSelector` action is given in Figure 6.2. In this specification, we see that the `CheckSelector` action is called recursively for rule processing after execution of `getAttr` methods. This recursion is only one level deep, because the `Sender` is always `This` in rule processing.

```
CheckSelector(Selector, Sender, ReplyTo) = {
   Event, Condition ← Selector
   if Sender ≠ This
   then {
      Allowed := ∅
      foreach Attr in Attr(Condition)
      where MethodAllowed(getAttr(), Sender)
      do Allowed := Allowed ∪ { Attr }

      if Allowed ≠ Attr(Condition)
      then Exit()

      foreach Attr in Attr(Condition)
      do {
         ExecuteMethod(getAttr(), Sender, Self)
         ⋆ ⋆ ⋆ Rule Processing ⋆ ⋆ ⋆
         ℛ = {R ∈ Rules | R = ⟨S, A⟩ ∧ CheckSelector(S, Self) ≠ ∅}
         ⟨S, A⟩ ← Pick random from ℛ
      }
   }

   Matching Set ← EventTest(Event)
   Evaluate Condition on Matching Set
   Return Matching Subhistories
}
```

Note: `Attr(Condition)` denotes the set of attributes occurring in `Condition`.

Figure 6.2: The algorithm executed by `CheckSelector`.

The use of `CheckSelector` reflects the two uses of selectors in DEGAS. Queries are sent to objects as calls to `CheckSelector`, which are queued in the query queue $\mathcal{Q}$. The selector of a rule is also checked using `CheckSelector`. These calls are made directly by the object itself, as is shown in Section 6.2.4.

MethodAllowed This action checks a given method call against the current lifecycle of the object, as defined in Definition 49. The input parameters are the sender of the call, the name of the method, and the input parameters of the call. `MethodAllowed` is a Boolean function, that can only be invoked from inside the object. Hence, its type is:

$$MethodAllowed : oid \times MethodCall \longrightarrow Boolean$$

Invocations of `MethodAllowed` are not recorded in the object's history.

ExecuteMethod This action executes a method. Its effect is given by the function M defined in Section 5.3. The effect of `ExecuteMethod` on the object is as defined in Definition 49. In a method, we can have three kinds of actions, viz., modifications of attributes, calls to methods within the object, and calls to methods in other objects. These are discussed in turn below.

We start with the effect of attribute modifications on the object itself. Given a state history $SH = SH(0);\ldots;SH(n)$ at time t, where

$$SH(n) = \langle t_n, \tau_n, I(\tau_n), m \rangle$$

The new state history SH' as a result of `ExecuteMethod` with a method call $\mu(q_1,\ldots,q_k)$ is:

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

Please note that `ExecuteMethod` itself is not recorded in the history.

Alternatively, a method can make calls to other methods. If the call MC is to a method of the object itself, then it is added to the internal method queue of the object:

$$\mathcal{M}'_{int} = \mathcal{M}_{int} + MC$$

where $\mathcal{M}'_{int}$ denotes $\mathcal{M}_{int}$ after execution of the call. Calls to methods in other objects are sent off to other objects using the `sendMessage` action provided by the system layer, that is discussed in Section 6.3. For a method call $Path.MC$, where $Path$ is a path expression evaluating to an object identity Obj, the following is executed:

`sendMessage(`Obj`,` MC`)`

The evaluation of path expressions in an object is translated to calls to `get` methods. The object awaits the answer to the `get` calls. Hence, this is a form of blocking communication. The answer to the `get` calls is delivered by a `sendReply` action of the system layer (see Section 6.3) in the reply box of the object. Assignment from the reply box $\mathcal{RB}$ blocks the object until the reply box is filled with a result. To prevent the object form blocking infinitely, a time-out is built in. If the block is timed out, the action is aborted and not recorded in the history.

`ExecuteMethod` is invoked from inside the object. It is not recorded in the object's history.

6.2.3 Standard Methods

The following methods are present in a DEGAS object. Like the primitive actions, they implement basic DEGAS functionality. Unlike the primitive actions, they are subject to the lifecycle specification. Since they are methods, they are recorded in an object's history.

`getPathExpr` This method evaluates a path expression given as parameter. If the path expression is an attribute of the object itself, then the appropriate `get` method is invoked. Otherwise, the head of the path expression is evaluated to serve as the destination of a `getPathExpr` message containing the tail of the path expression. As a consequence, the evaluation of a path expression requires a sequence of messages between objects. An example is shown in Figure 6.3. There, we see the use of the `replyTo` field of a message. Access to the successive attributes in the path expression is dependent on the access rights of the previous object in the evaluation chain, that is the `sender` of the method. The answer, however, must be returned to the original sender of the complete path expression, object A in Figure 6.3. This is achieved by including its identity as the `ReplyTo` object. The final object in the evaluation chain, object Z in Figure 6.3, executes a `return` statement in its `getAm` method, which returns the value of `Am` to the `ReplyTo` object, i.e., object A.

The algorithm executed by `getPathExpr` method is the following:

```
getPathExpr(pathexpr) = {
   if ∃ attribute ∈ Attr: pathexpr = attribute
   then
      M_ext := M_ext + ⟨ sender=Sender, replyTo=ReplyTo, getattribute ⟩
   else
      sendMessage(Dest=head(pathexpr), sender=Sender,
                  replyTo=ReplyTo, Mesg=getPathExpr(pathExpr))
   fi
}
```

`get` A `get` action is defined for every attribute of the object. For the attribute `Attr`, the associated actions is named `getAttr`. It is used to specify access to attributes from outside. The attributes of a DEGAS object are accessible from outside in two manners: queries and path expressions. In both cases, a reference to an attribute is treated as a call to the associated `get` method. Since it is a method, every call to `getAttr` is entered in the history of the DEGAS object.

For a query, the `sender` of the `get` method is set to the sender of the query. For example, suppose the following query on the class `Employee` is issue by the object *Taxman*:

```
Select from Employee
if Salary ≥ 70000
```

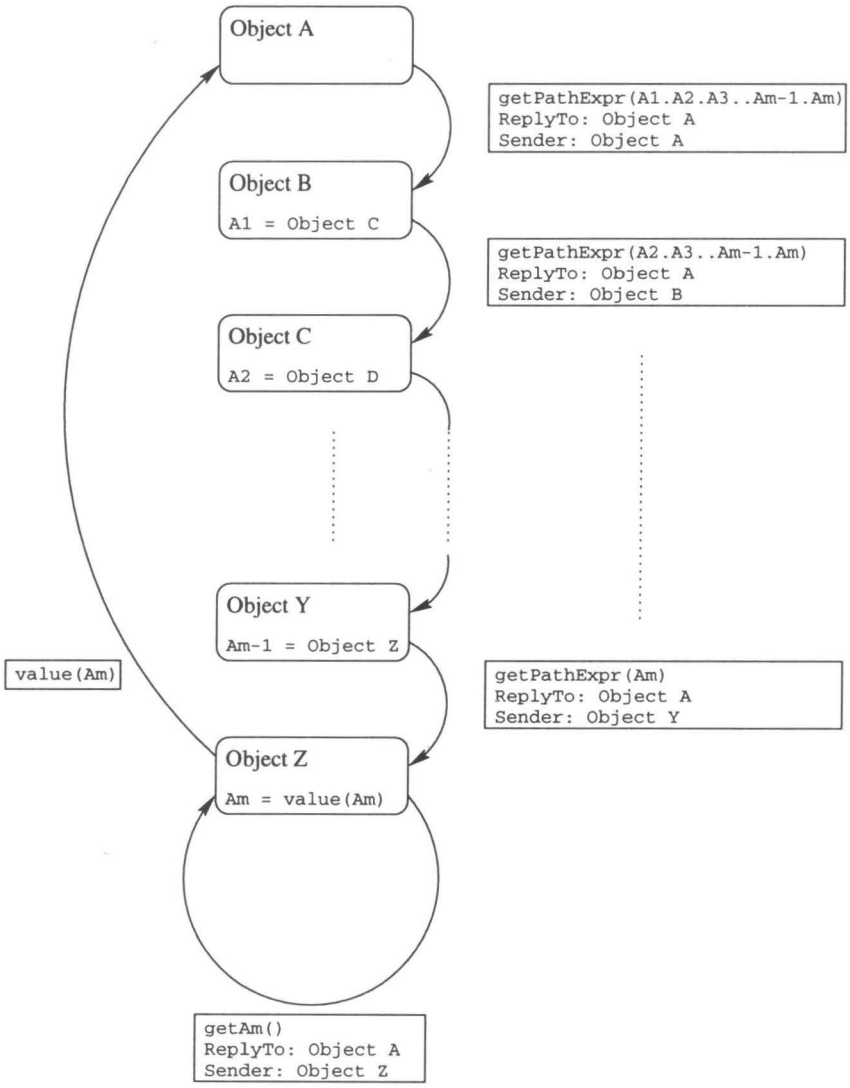


Figure 6.3: Evaluation of a DEGAS path expression.

To check this condition `CheckSelector` executes the method `getSalary` with $sender = Taxman$. Hence, the lifecycle checks whether $Taxman$ has access to the attribute `Salary`. Likewise, the evaluation of path expressions also leads to calls to `get`. The evaluation of a path expression is discussed above in the specification of `getPathExpr`.

The default lifecycle of a `get` method is to allow everyone to call it. Hence, if the programmer does not specify a lifecycle for the `getSalary` method, then it is:

Lifecycle
getSalary*

If we wish to restrict the access to an attribute, we specify a guard for this method. For example, the following lifecycle restricts access to `salary` to the tax inspector.

Lifecycle
[sender=Taxman]getSalary*

Of course, any other restriction on the execution of `getSalary` is possible. The composition of lifecycles using the communication merge operator |, discussed in Section 5.7.2, ensures that these specifications are orthogonal to the rest of the lifecycle specification.

`Extend` This action extends the object with an addon. The name of the addon is given as a parameter. Given a call to `Extend` with addon $\mathcal{A}$ as a parameter. The capabilities defined by $\mathcal{A}$ are given by the following sets:

$$Attr(\mathcal{A}), Meth(\mathcal{A}), Cycl(\mathcal{A}), Rules(\mathcal{A})$$

The effect on the object of executing `Extend(`$\mathcal{A}$`)` is given by

$$\begin{aligned} Attr' &= Attr \cup Attr(\mathcal{A}) \\ Meth' &= Meth \cup Meth(\mathcal{A}) \\ Cycl' &= Cycl \cup Cycl(\mathcal{A}) \\ Rules' &= Rules \cup Rules(\mathcal{A}) \end{aligned}$$

where $Attr'$ denotes the set of attributes after execution of `Extend`. The meaning of $Meth'$, $Cycl'$, and $Rules'$ is similar. Please note, that we assume unique names for attributes and methods, as stated in Section 4.3.

The lifecycle of the object after execution, denoted by LC', is:

$$LC' = |_{\lambda \in Cycl'} \lambda$$

where | denotes communication merge with communication function $\gamma(\mu, \mu) = \mu$ for all $\mu \in Meth'$.

To implement the extension with an addon, an object requests the appropriate schema information from the addon class object. The capabilities in the addon are then added to the capabilities of the object. The addon class object is referred to through the name of the addon. The actions of the addon class object are specified in Section 6.5.

Remove This action removes an addon from an object. The name of the addon is given as a parameter. Given a call to `Remove` with addon $\mathcal{A}$ as a parameter. The capabilities defined by $\mathcal{A}$ are given by the following sets:

$$Attr(\mathcal{A}), Meth(\mathcal{A}), Cycl(\mathcal{A}), Rules(\mathcal{A})$$

The effect on the object of executing `Remove`($\mathcal{A}$) is given by

$$\begin{aligned} Attr' &= Attr \setminus Attr(\mathcal{A}) \\ Meth' &= Meth \setminus Meth(\mathcal{A}) \\ Cycl' &= Cycl \setminus Cycl(\mathcal{A}) \\ Rules' &= Rules \setminus Rules(\mathcal{A}) \end{aligned}$$

where $Attr'$ denotes the set of attributes after execution of `Remove`. The meaning of $Meth'$, $Cycl'$, and $Rules'$ is similar. The lifecycle of the object after execution, denoted by LC', is:

$$LC' = |_{\lambda \in Cycl'} \lambda$$

where $|$ denotes communication merge with communication function $\gamma(\mu, \mu) = \mu$ for all $\mu \in Meth'$.

Kill The `Kill` action terminates the existence of an object. By default, it can only be executed by the class object and the object itself. Hence, its default lifecycle specification is:

Lifecycle
[sender=ClassObject **or** Sender=Self] Kill

Default Lifecycle In the specification of a DEGAS objects' actions, we referred a number of times to the default lifecycle of a DEGAS object. Actions that can be invoked from outside and that are recorded in the object's history, must be specified in the lifecyle of an object. This is the case for the following actions: `get`, `Extend`, `Remove`, and `Kill`.

Lifecycle
⋆ ⋆ ⋆ For each `Attribute` ∈ $Attr$: ⋆ ⋆ ⋆
get<Attribute>*
Extend*
Remove*
[Sender=ClassObject **or** Sender=This] Kill

6.2.4 Execution

The functionality of a DEGAS object is implemented by an execution cycle that uses the actions defined in the previous subsection. Basically, the object cycles through two activities, viz., processing queries and executing method calls. Rule execution is done as part of method execution. The execution cycle is depicted in Figure 6.4.

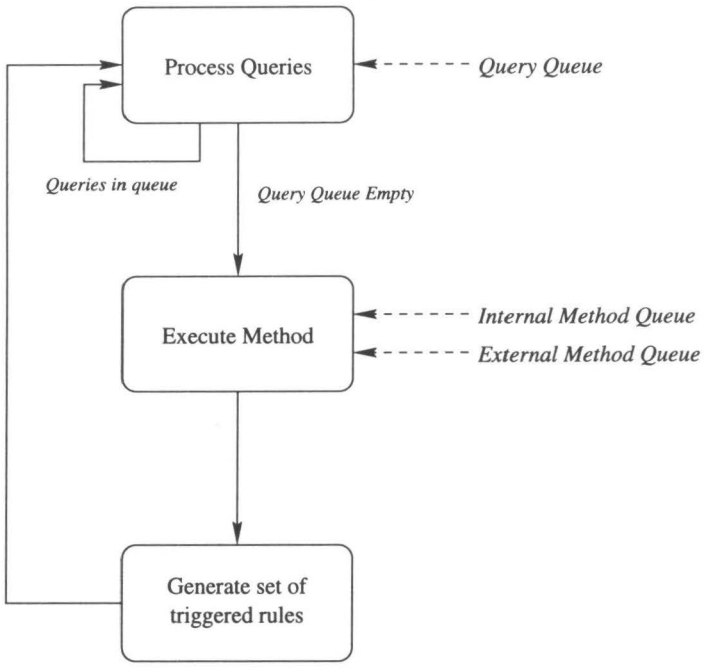


Figure 6.4: Execution Cycle of a DEGAS Object

The exact actions of the object are given in the algorithm in Figure 6.5. In the query processing stage, the object process the complete query queue $\mathcal{Q}$. If $\mathcal{Q}$ is empty, then it proceeds with method execution. Here, method calls from inside the object take precedence over method calls from outside. Hence, the object only takes a method call MC from $\mathcal{M}_{ext}$, if $\mathcal{M}_{int}$ is empty. The call MC is then checked against the lifecycle of the object. If it is not allowed it is discarded. It MC is allowed, then MC is executed.

After execution of a method, the set of triggered rules is generated. One rule is then picked at random for execution. Execution of the rule's action means that the call is made. In the case of an internal method call, it is appended to $\mathcal{M}_{int}$. If it is a call to another object, it is sent to that object.

```
⋆ ⋆ ⋆ Execution Cycle of a DEGAS object ⋆ ⋆ ⋆
Repeat
   ⋆ ⋆ ⋆ Query Processing ⋆ ⋆ ⋆
   While Q not empty do
      ⟨Sender, ReplyTo, QueryID, Selector⟩ ← Head(Q)
      Result := CheckSelector(Selector, Sender)
      ReplyTo.queryResult(QueryID, Result)
   od

   ⋆ ⋆ ⋆ Method Execution ⋆ ⋆ ⋆
   Repeat
      if M_int not empty
         Sender, ReplyTo, MC ← Head(M_int)
         M_int ← tail(M_int)
      else
         Sender, ReplyTo, MC ← Head(M_ext)
         M_ext ← tail(M_ext)
      fi
   Until MethAllowed(Sender,MC)
   ExecuteMethod(MC)
   Return answer to ReplyTo, if necessary

   ⋆ ⋆ ⋆ Rule Processing ⋆ ⋆ ⋆
   R = {R ∈ Rules | R = ⟨S, A⟩ ∧ CheckSelector(S, Self) ≠ ∅}
   ⟨S, A⟩ ← Pick random from R
   ⋆ ⋆ ⋆ The action is executed ⋆ ⋆ ⋆
   Send message A
Until The End of this Object
```

Figure 6.5: The Execution of a DEGAS Object

Action of the object	Event History EH	Internal Method Queue $\mathcal{M}_{int}$
Execute μ_1	μ_1	$[\mu_2, \mu_3]$
R_1 triggered		$[\mu_2, \mu_3, \alpha_1]$
Query processing		
Execute μ_2	$\mu_1; \mu_2$	$[\mu_3, \alpha_1]$
R_2 triggered		$[\mu_3, \alpha_1, \alpha_2]$
Query processing		
Execute μ_3	$\mu_1; \mu_2; \mu_3$	$[\alpha_1, \alpha_2]$
Query processing		
Execute α_1	$\mu_1; \mu_2; \mu_3; \alpha_1$	$[\alpha_2]$
Query processing		
Execute α_2	$\mu_1; \mu_2; \mu_3; \alpha_1; \alpha_2$	[]

Figure 6.6: An example execution by a DEGAS object

As an illustration of this algorithm, we show an example execution of a DEGAS object O. In this example, we abstract from the data in O in order to focus on the dynamic aspects. The object has five methods, μ_1, μ_2, μ_3, α_1, and α_2. All methods manipulate data except μ_1, which calls μ_2 and μ_3. The lifecycle of O is:

$$((\mu_1 \| (\mu_2; \mu_3)); (\alpha_1 \| \alpha_2))^*$$

Furthermore, O has two rules:

R_1 = **On** μ_1
do α_1

R_2 = **On** μ_2
do α_2

We show a short snapshot of O's execution in Figure 6.6. It starts with the execution of μ_1. We do not show the checks against the lifecycle, since it is obvious that this execution satisfies O's lifecycle.

In DEGAS rule processing, non-executed triggered rules are discarded by the object. In combination with the negation operator, this gives the application programmer a great degree of flexibility in the interaction of rules and lifecycles. If the action of a rule is not executed, we have two options, viz., to leave the rule unexecuted or to retry it at a later time. A rule is left unexecuted in situations where only a timely reaction is useful. An example is found in automatic trading in financial markets. There, our reaction to a falling price of shares might be 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.

A rule is retried, if the action must always be executed once a rule has been triggered. Rules that maintain integrity constraints will use such a strategy.

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

On $A;B$
do μ

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 μ

This rule is triggered, as long as no μ action is execute after the occurrence of $A;B$.

We can also characterise the DEGAS execution model using the dimensions of rule execution in active databases, discussed in Section 3.1. Since rules are encapsulated in an object, and an object executes as a separate thread, rule execution in DEGAS is instance-oriented. Furthermore, the coupling of event and condition is immediate, since they are checked as a unit by `CheckSelector`. The action of a rule is queued into the internal method queue $\mathcal{M}_{int}$. Hence, condition - action coupling is deferred.

6.3 System Layer

The basic DEGAS object is built on top of a system layer. It provides the lowest level implementation of objects, i.e., an abstraction of the physical object level. The two functions implemented by the system layer are creation of new objects and communication between objects.

Empty DEGAS objects created by the system layer implement the functionality defined in Section 6.2, but does not contain any capabilities. Hence, a class object has to add these in order to make the object an instance of its class.

Communication between objects in DEGAS can take place in two ways:

1. **Point-to-point asynchronous.** This kind of communication is through method calls. These are sent off by an object to a specified other object without expecting an answer.

2. **Point-to-point synchronous.** This kind of communication is used to evaluate path expressions. The object evaluating the expression awaits the result from another object.

3. **Broadcast.** The broadcast facility offered by the DEGAS system layer sends a message to all objects in a class. An example of its use is in query processing, as discussed in Section 7.2.

The communication primitives defined in this section act directly on the data structures specified for the basic DEGAS object.

The recipient of a message is specified by its identity. The point-to-point communications action additionally accept names as destination objects. The name is resolved to an object identity by the system layer.

In the following definition of the system layer, we consider this layer as one entity. In Chapter 7, we discuss the implementation of the system layer in a distributed environment.

6.3.1 Data Structures

The system layer records a directory of all objects and their addresses.

`Object Directory` This contains all objects in the DEGAS database with their physical addresses. It is denoted by $ObjDir$. The pseudo-type of $ObjDir$ is:

$$ObjDir : \mathcal{P}(Oid \times PhysAddr)$$

$ObjDir$ is only used internally by the system layer to deliver messages to DEGAS objects.

`Name Directory` This directory records the identities of named objects. It maps names to object identities. It is specified as follows:

$$NameDir : \mathcal{P}(String \times Oid)$$

$NameDir$ is used internally by the system layer to support communication to named objects. Objects recorded in `NameDir` can be specified by name as a destination. $NameDir$ contains the following objects:

1. Class objects
2. Relation Class objects
3. Addon Class objects
4. A site object

The former three categories are referenced by the name of the class with `Class` appended. For example, the class object of `Person` is known as `PersonClass`. The latter is referenced by the name `Site`.

6.3.2 Actions

Here, we specify the actions offered by the system layer in terms of their effects on the data in the system layer and the objects involved.

NewEmptyObject This action creates a new empty DEGAS object. A call to NewEmptyObject results in an empty DEGAS object with identity $NewOid$ at physical address $NewAddr$. Hence, the type of NewEmptyObject is:

$$NewEmptyObject : \mathbf{1} \rightarrow Oid \times PhysAddr$$

The new object directory $ObjDir'$ after execution of this action is:

$$ObjDir' = ObjDir \cup \{NewOid \times NewAddr\}$$

The action NewEmptyObject can be invoked by class objects. $NewOid$, the identity of the new object, is returned to the sender of the action. Its use by the class object is explained in Section 6.4.

sendMessage This action implements asynchronous point-to-point communication. It sends a message from an object to another object. Messages are always method calls in DEGAS. The parameters of sendMessage are the following:

Destination object	$Dest$	Oid
Sending object	$Sender$	Oid
Recipient of reply	$replyTo$	Oid
Message	$Mesg$	$MethodCall$

The destination of sendMessage can also be specified by a name. The effect of sendMessage is that the message $Mesg$ is added to the external method queue $\mathcal{M}_{ext}$ of $Dest$. The external method queue after execution is denoted by $\mathcal{M}'_{ext}$:

$$Dest.\mathcal{M}'_{ext} = Dest.\mathcal{M}_{ext} + Mesg$$

Any object can invoke sendMessage.

sendQuery This action implements another form of asynchronous point-to-point communication. It sends a query from an object to another object. The parameters of sendQuery are the following:

Destination object	$Dest$	Oid
Sending object	$Sender$	Oid
Recipient of reply	$replyTo$	Oid
Query Identity	$QueryID$	$integer$
Selector	Sel	$SelectorType$

The destination of `sendQuery` can also be specified by a name. The effect of invocation of `sendQuery` by an object O_1 on behalf of an object O_2 is that a tuple $Query$ added to the query queue $\mathcal{Q}$ of $Dest$, where

$$Query = \langle sender = O_1, replyTo = O_2, qid = QueryID, Selector = Sel \rangle$$

The query queue after execution is denoted by $\mathcal{Q}'$:

$$Dest.\mathcal{Q}' = Dest.\mathcal{Q} + Query$$

Any object can invoke `sendQuery`.

`sendReply` This action implements synchronous point-to-point communication. It is used to return a value to an object. The parameters of `sendReply` are the following:

Destination object	$Dest$	Oid
Reply	$Answer$	value

The destination of `sendReply` can also be specified by a name. The effect of `sendReply` is that the value $Answer$ is put into the reply box $\mathcal{RB}$ of $Dest$. The reply box after execution is denoted by $\mathcal{RB}'$:

$$\mathcal{RB}' = Answer$$

`Broadcast` This action sends a message to all objects in a class. The parameters of `Broadcast` are a class name $DestClass$ and a method call $Mesg$:

Destination class	$DestClass$	$Classname$
Sending object	$Sender$	Oid
Recipient of reply	$replyTo$	Oid
Message	$Mesg$	$MethodCall$

Execution of `Broadcast` effects the external method queues $\mathcal{M}_{ext}$ of all objects in class $DestClass$. The external method queue $\mathcal{M}'_{ext}$ after execution is as follows:

$$\forall O \in DestClass : O.\mathcal{M}'_{ext} = O.\mathcal{M}_{ext} + Mesg$$

Any object can invoke `Broadcast`.

6.3.3 Execution

The execution of the system layer is completely driven by requests from DEGAS objects. Calls to actions in the system layer are executed on a First Come, First Served basis. This is the only place in DEGAS, where object autonomy is compromised by introducing a form of synchronisation. Clearly, this is necessary to allow objects to communicate.

6.4 Class Objects

Class objects implement a major part of object management. They are responsible for the creation of new object instances of their class. As a consequence, a class object keeps a record of existing instances. Furthermore, a class object provides schema information.

6.4.1 Data Structures

The following data is recorded in a class object, in addition to the data recorded as standard in a DEGAS object.

Extent This set contains the identities of the objects in the extent of the class managed by the class object. Because the system layer provides an abstraction from physical addresses, the type of $Extent$ is:

$$Extent : \mathcal{P}OId$$

Class Attribute Set This set contains the attributes specified for the class. It is denoted by $ClassAttr$.

Class Method Set This set contains the methods specified for the class. It is denoted by $ClassMeth$.

Class Lifecycle Set This set contains the lifecycles specified for the class. It is denoted by $ClassCycl$.

Class Rule Set This set contains the rules specified for the class. It is denoted by $ClassRules$.

6.4.2 Actions

A class object provides all actions of a DEGAS object. In addition, it provides actions to create new objects and actions to provide schema information.

New This action produces a new object in the class. First, a new empty DEGAS object is requested form the system layer. Then, this object is filled with the capabilities specified for the class. The created object is added to the extent of the class.

After the first phase, the following conditions hold:

$$\begin{aligned} NewOid &= \texttt{NewEmptyObject()} \\ NewOid.Attr &= \emptyset \end{aligned}$$

$$\begin{aligned} NewOid.Meth &= \emptyset \\ NewOid.Cycl &= \emptyset \\ NewOid.Rules &= \emptyset \end{aligned}$$

Filling the object means that the following post-conditions are satisfied, where $Extent'$ denotes the value of $Extent$ after execution of New.

$$\begin{aligned} Extent' &= Extent \cup \{NewOid\} \\ NewOid.Attr &= ClassAttr \\ NewOid.Meth &= ClassMeth \\ NewOid.Cycl &= ClassCycl \\ NewOid.Rules &= ClassRules \end{aligned}$$

getExtent This action returns the extent managed by the class object.

$$getExtent : \mathbf{1} \longrightarrow \mathcal{P}Oid$$

This action can be invoked by a `Site` object.

IsAttribute This is a Boolean function to check for the presence of an attribute in a class. `IsAttribute` is used to check the correctness of queries, as is discussed in Section 7.2. As parameters, it takes the name and type of an attribute. It is typed:

$$IsAttribute : String \times Type \longrightarrow Boolean$$

The `IsAttribute` action can be called by `Site` objects.

IsMethod This is a Boolean function to check for the presence of a method in a class. `IsMethod` is used for type checking. As parameters, it takes the name of the method and its parameters. It is typed:

$$IsMethod : String \times [\langle name : string, Type \rangle] \longrightarrow Boolean$$

The `IsMethod` action can be called by `Site` objects.

6.5 Addon Class Object

Information about addons is stored in addon class objects. An addon class object is similar to a class object. The only difference is the absence of an action to create a new object. Since an addon extending an object does not have object identity, an addon class object only adds and removes existing object identifiers to and from its extent. The class extended by the addon, specified in the `Extends` clause in the addon definition, is denoted by $BaseClass$.

```
Object PersonClass
Attributes
   Extent :POid
   ClassAttr : P⟨ Name, Type ⟩
   ClassMeth : P⟨ Name, Type ⟩
   ClassCycl
   ClassRules
   NewOid : Oid
   attr : ⟨ Name, Type ⟩
   meth : ⟨ Name, Type ⟩
   site : oid
Methods
   New = {
      NewOid := NewEmptyObject()
      Extent : = Extent ∪ {NewOid}
      NewOid.Attr := ClassAttr
      NewOid.Meth := ClassMeth
      NewOid.Cycl := ClassCycl
      NewOid.Rules := ClassRules
      Return NewOid
   }
   getExtent = {
      return Extent
   }
   IsAttribute(Name, Type) = {
      Foreach attr in ClassAttr
      where attr.name =Name and attr.type = Type
      do return True
   }
   IsMethod(Name, Type) = {
      Foreach meth in ClassAttr
      where attr.name =Name and attr.type = Type
      do return True
   }
Lifecycle
   New*
   [sender=site]getExtent*
   [sender=site]isAttribute*
   [sender=site]isMethod*
Rules
EndObject
```

Figure 6.7: An example DEGAS class object.

6.5.1 Data Structures

The following data is recorded in an addon class object, in addition to the data recorded as standard in a DEGAS object.

Extent This set contains the identities of the objects extended with the addon managed by the addon class object. Hence, it has a slightly different meaning than *Extent* in a class object. The type of *Extent* is:

$$Extent : \mathcal{P}Oid$$

Addon Attribute Set This set contains the attributes specified for the addon. It is denoted by *AddonAttr*.

Addon Method Set This set contains the methods specified for the addon. It is denoted by *AddonMeth*.

Addon Lifecycle Set This set contains the lifecycles specified for the addon. It is denoted by *AddonCycl*.

Addon Rule Set This set contains the rules specified for the addon. It is denoted by *AddonRules*.

6.5.2 Actions

The actions of an addon class object are to provide information about the capabilities of an object.

getExtent This action returns the objects extended by the addon of the addon class object, i.e., the value of *Extent*.

$$getExtent : \mathbf{1} \rightarrow \mathcal{P}Oid$$

This action can be invoked by a site object.

registerExtent This action registers the extension of an object with the addon. It does not have any parameters, because the message is sent by the extended object. The execution of `registerExtent` results in the addition of the sender's identity to *Extent*. *Extent'* denotes the extent of the addon class after execution:

$$Extent' = Extent \cup \{Sender\}$$

The action `registerExtent` can be called by objects in *BaseClass*.

removeExtent This action informs the addon class object of the removal of an addon. It does not take any parameters, because the message is sent by the object the addon is removed from. The execution of `removeExtent` results in the removal of the sender's identity from $Extent$. $Extent'$ denotes the extent of the addon class after execution:

$$Extent' = Extent \setminus \{Sender\}$$

The action `removeExtent` can be called by objects in $BaseClass$.

IsAttribute This is a Boolean function to check for the presence of an attribute in a class. `IsAttribute` is used to check the correctness of queries, as is discussed in Section 7.2. As parameters, it takes the name and type of an attribute. It is typed:

$$IsAttribute : String \times Type \rightarrow Boolean$$

The `IsAttribute` action can be called by `Site` objects.

IsMethod This is a Boolean function to check for the presence of a method in a class. `IsMethod` is used for type checking. As parameters, it takes the name of the method and its parameters. It is typed:

$$IsMethod : String \times [\langle name : string, Type \rangle] \rightarrow Boolean$$

The `IsMethod` action can be called by `Site` objects.

Default Lifecycle The default lifecycle of an addon class object ensures that the actions to extend an object are executed in the right sequence. In particular, this lifecycle guarantees that an object can only register with the addon, if it has executed the necessary action for extension.

Lifecycle
[sender $\in$ $BaseClass$]registerExtent*
[sender $\in$ $Extent$]removeExtent*
[sender = site]getExtent*

6.6 Relation Objects

A relation object is also a DEGAS object. Hence, it contains all data structures and actions defined in Section 6.2. In addition, a relation object always has attributes to record the identities of the partners in the relation. Furthermore, the involvement of partners means that terminating a relation is more complex than simply killing the relation object.

6.6.1 Data Structures

Additional attributes provided by the relation object contain the identities of the partners in the relation.

Partners The specification of a relation object specifies the partners in the relation in the `Relation` clause. For example, consider the `Relation` clause in a `Share` relation:

Object Share
Relation Company, Shareholder
⋮

As a result the `Share` relation object contains the attributes `Shareholder` and `Company`. Furthermore, the partners in the relation are recorded as a set in the attribute `Partners`. In general, consider a relation object R with the following header:

Object R
Relation $P_1, P_2, \ldots, P_n$
⋮

Then we have the following elements in the attribute set of R:

$$\forall 1 \leq i \leq n : P_i : oid \in Attr$$
$$Partners : \mathcal{P}Oid \in Attr$$

The partner objects in the relation are stored a second time in a set to be able to send a message to all partner object at the same time through a set iteration.

6.6.2 Actions

An action to terminate the relation is provided in addition to the straight `kill` in a DEGAS object.

Terminate This action implements termination of the relation. As a result of `Terminate` the relation object ceases to exist. Furthermore, the partners in the relation must be informed of the end of the relation.

```
...
Methods
  ...
  Terminate = {
    ⋆ ⋆ ⋆ For each Part ∈ Partners ⋆ ⋆⋆
    Part.terminate<Part>()
  }
Rules
  On Terminate do Kill
  ...
```

6.7 Relation Class Objects

A relation class object is a class object. Hence, it contains all capabilities of a class object and, by implication, all capabilities of a DEGAS object. The main role of a relation class object is to match partners for a relation. The way this takes place is dependent on the application semantics of the relation. Hence, this section only gives a standard interface for initiating relations. Possible scenarios for use in applications are discussed in Section 7.3.

6.7.1 Data Structures

The additional data structures in a relation class object relative to a class object serve to record data about prospective partners in relations that are being formed.

`Prospect` This attribute records prospective partners in a relation. It is denoted by $Prospect$. Since the result of the matching process are relations, the prospective partners are recorded as tuples of objects. The set of partners in a relation is denoted by $Partners$.

$$\forall \texttt{Part} \in Partners : Prospect : \mathcal{P}\langle \texttt{<Part>} : oid \rangle$$

A tuple in this set denotes a potential combination of objects to form a relation. The missing partners in a combination of prospects are represented by a `Null` value in the tuple. For example, suppose we have a three way relation `Schedule` with partners `Teacher`, `Course`, and `Room`. An example `Prospect` set in the relation class object is:

$$\{\langle Teacher : 123, Course : 345, Room : Null\rangle,$$
$$\langle Teacher : 135, Course : 368, Room : Null\rangle,$$
$$\langle Teacher : Null, Course : 369, Room : 981\rangle\}$$

This relation class object has two teacher - course pairs looking for a room and one course - room pair lacking a teacher.

6.7.2 Actions

The following actions are defined in addition to the actions of a class object.

`initiate` This action is used by prospective partners to express interest in engaging in a relation. There is an `initiate` action for each partner in the relation. Let $Meth$ be the method set of a relation class object RC, then:

$$\forall \texttt{Part} \in Partners : initiate\texttt{<Part>} \in RC.Meth$$

In the example of the `Schedule` relation class object, we have the following actions:

```
initiateTeacher
initiateCourse
initiateRoom
```

An `initiate` method can have parameters, for example containing the desired partner in the relation. The exact actions of these methods are dependent on the application, as will be discussed in Section 7.3. As a consequence, the relation class object is specified by the application programmer.

`instantiateRelation` This action does the actual instantiation of the relation. This means that it creates the relation object and instructs the partners to extend themselves with the appropriate addon. Furthermore, it removes the tuple of partners from `Prospects`.

Methods
⋆ ⋆ ⋆ ∀Part ∈ *Partners* : ⋆ ⋆ ⋆
instantiateRelation(<Part> : oid) = {
  Part.extend(<Part>)
  Relation := new()
  Part.initialise(Relation)
  Foreach p **in** Prospects
  where p.<Part> = <Part>
  do Prospects := Prospects - p
}

This action can only be invoked by the object itself.

Default Lifecycle The default lifecycle of the `initiate` actions allows their execution at any time.

Lifecycle
⋆ ⋆ ⋆ For each <Part> ∈ *Partners* ⋆ ⋆⋆
Initiate<Part>*
[Sender=Self] instantiateRelation*

6.8 Site Objects

A `Site` object facilitates object management, as discussed in Section 4.5. It has a number of class objects, that provide schema information and record the local extent of their class. The main activity of a `Site` object is related to query processing.

6.8.1 Data structures

Data stored by the `Site` object are the classes known by the site and intermediate results of queries.

ClassesOnSite This attribute contains the set of classes on the site. For each class object, the Site object records name and identity.

$$ClassesOnSite : \mathcal{P}\langle ClassName : String, ClassObj : Oid\rangle$$

The *local extent* of the class, i.e., the part of the class extent at this site, is stored by the class object, as discussed in Section 6.4

QResults This attribute contains the results of queries processed by the Site object. The result of a single query is a set of object - history pairs. Its type is denoted by $QueryResultType$

$$QueryResultType = \mathcal{P}\langle Object : Oid, EH : EHist\rangle$$

Please note that $EHist$ denotes the type of an event history, as defined in Definition 28.

Because a Site object must have the ability to process multiple queries at a time, it also records a query identity and the query generating object.

$$QResults : \mathcal{P}\langle Qid : integer, Issuer : Oid, Result : QueryResultType\rangle$$

The query identity serves to distinguish multiple queries processed at the same time by the Site object. It is also sent with the CheckSelector request, as we saw in Section 6.2.

NextQid This is a numerical attribute containing the next query identity to be handed out:

$$NextQid : integer$$

It is simply a number, that is increased each time a query is distributed by the Site object.

6.8.2 Actions

The actions of a Site object are mainly concerned with query processing. It offers facilities to distribute a query over a class, collect the results from the instances, and ship the result back to the object issuing the query.

DistributeQuery This action is used to distribute a query over the instances of a class. As parameters it takes the identity of the query, the name of the class and the query to be distributed.

$$Class : ClassName$$
$$Query : \langle sender : oid, Selector : SelectorType\rangle$$

To distribute the query, the `site` object first assigns an identity to the query. Then, it obtains the local extent of the class from the class object. After that, the `Site` object sends the query to all objects in the extent using the `sendQuery` primitive of the system layer (see Section 6.3).

```
distributeQuery(Class : String, Query : QueryType ) = {
   if not Class.isAttribute(Attributes in Selector)
         or not Class.isAttribute(Events in Selector)
   then exit

   qid := NextQid
   NextQid := NextQid + 1
   extent := Class.getExtent()
   foreach o in extent
   do sendQuery(dest=o, sender=Query.Sender, replyTo=This,
                     QueryID= qid, Sel = Query.Selector)
}
```

In this specification `QueryType` is shorthand for the type of a query, as given above.

`queryResult` This action is used to collect the results of a query. It adds a given result for an individual object to the set of query results $QResults$ stored in the object. The parameters of `queryResult` are the identity of the query and the set of matching sub-histories in the object history.

$$Qid : integer$$
$$LocalResult : \mathcal{P}EHist$$

The effect of this action is that the local result is added to the set of query results.

$$\begin{array}{l} QR \in QResults: \\ \quad QR.Qid \;=\; Qid \\ \quad QR.Result' \;=\; QR.Result \cup \\ \qquad \{\langle object = ObjectID, EH = hist\rangle \mid \\ \qquad\; ObjectID = Sender \wedge hist \in LocalResult\} \end{array}$$

The action `queryResult` can be invoked by any object.

`shipResult` This action sends the result of a query to the object issuing the query. The result of a query q is taken from $QResults$. After it is shipped, the result of q is removed from $QResults$. It's actions are specified as follows in DEGAS:

```
shipResult(Qid : integer) = {
   Foreach qr in QResult
   where qr.qid = Qid
   do {
```

```
            sendMessage(qr.Issuer, This, This, answerQuery(qr.Result))
            QResults := QResults - qr
        }
    }
```

The action `shipResult` can only be invoked by the `Site` object itself.

Default Lifecycle The lifecycle of a `Site` object prescribes the correct sequence for query processing. First, the object distributes the query. Then, it receives the answers from the object instances. Finally, it ships the result back to the sender of the query:

Lifecycle
| (distributeQuery;queryResult*;shipResult)*

6.9 Conclusion

In this chapter, we gave a functional specification of DEGAS as an intermediate step between the abstract semantics and an implementation. To that end, we identified for each element needed to implement a DEGAS database, what data is stored and what actions are required.

As a foundation, a system layer offers communication and object creation services. These are necessary for the basic DEGAS object, which implements all object capabilities, viz., attributes, methods, lifecycles, and rules, using a number of primitive actions. These actions are executed as part of a cycle, that processes query requests, then executes a method, and processes triggered rules.

All further elements of a DEGAS database are specified as DEGAS objects themselves. For these objects, we specified the actions required for a DEGAS database. Relation objects must offer actions to terminate the relation. Class objects, relation class objects, and addon class objects all take care of creating instances in their class. The messages to handle this are standardised. Site objects do not implement part of the data model. Instead, their main task is to facilitate query processing

Chapter 7

Practical Aspects of DEGAS

This chapter addresses a number of issues in the realisation of a DEGAS database. First, we discuss the prototype of a DEGAS implementation. In the introduction of Chapter 6, we positioned the functional specification as half way between the abstract semantics of DEGAS, specified in Chapter 5, and a DEGAS implementation. Hence, this chapter discusses the implementation of the system specified in the previous chapter. Additionally, we shortly discuss the interface of DEGAS to the outside world.

Furthermore, the specification given in Chapter 6 gave only a specification of a standard interface for creating new objects and new relation objects. In this chapter, we discuss how application dependent semantics can be programmed in DEGAS using this standard interface. Furthermore, we discuss how the actions specified in Chapter 6 are used to implement query processing in DEGAS. This discussion includes the maintenance of a data dictionary and the approximation of query result quality in the context of DEGAS object autonomy.

7.1 Implementation of DEGAS

The implementation of a DEGAS database is explained by showing how to implement each element of the functional specification. For this discussion, we draw on our experience with the implementation of an early DEGAS prototype in Python. In this section, we first motivate our choice for Python as the implementation language. Then, we explain the implementation of the key elements of a DEGAS database. These are the basic DEGAS object and the system layer. Other elements of a DEGAS database are themselves DEGAS objects. Therefore, we can implement these, if we can implement a basic DEGAS object.

7.1.1 The Implementation Language

The aim of the prototype DEGAS implementation was to provide a proof of concept for the model defined in this thesis. The prototype is meant to show that a DEGAS database can be implemented. This lead to the following requirements on the implementation language:

1. **Object support.** Besides the obvious advantages of object orientation in software development, the presence of objects makes implementation an object-based system like DEGAS easier.

2. **Facility for threads or processes.** DEGAS objects run concurrently. Hence, the implementation platform must support concurrency.

3. **Made for Prototyping.** Since the prototype is only meant to provide a proof-of-concept, quick implementation is more important than optimal performance.

We chose Python [Lutz, 1996] as the implementation platform for a DEGAS prototype. Python[1] is an object-oriented scripting language developed at CWI [Rossum, 1995b, Rossum, 1995c]. It is especially suitable for rapid prototyping, since it draws on earlier experience with ABC [Geurts *et al.*, 1990], designed from the viewpoint of a programming language as a user interface. Python has a number of features that were of particular use in writing a DEGAS prototype. It has a large number of built-in data structures, such as list and dictionaries. A Python dictionary has the usual structure of a key followed by an entry. As an example, suppose we enter the following in the Python interpreter:

```
>>> telefoon = {}
>>> telefoon['Johan'] = 4134
>>> telefoon['Arno'] = 4139
>>> telefoon['Arjan'] = 4054
```

Then, the contents of `telefoon` are as follows:

```
>>> telefoon
{'Arjan': 4054, 'Johan': 4134, 'Arno': 4139}
```

Elements are deleted from a dictionary using the `del` statement:

```
>>> del telefoon['Arjan']
>>> telefoon
{'Johan': 4134, 'Arno': 4139}
```

A dictionary offers a structure of variable size and content to store a DEGAS object's capability sets. For example, the attribute values of an object are stored in a dictionary indexed by name. This allows easy addition and deletion of attributes, as is discussed in Section 7.1.2.

Furthermore, Python allows references to functions by name. This is a very useful feature for the implementation of the addon mechanism. To illustrate this, suppose we have defined the following function in the Python interpreter:

[1] An extensive source of information on Python is the web site `www.python.org`.

```
>>> def multiply(parameter):
...     number = parameter * 2
...     return number
```

Then we assign:

```
>>> naam = multiply
```

Calling `naam` leads to the execution of `multiply`:

```
>>> naam(45)
90
```

This feature, together with dictionaries, facilitates easy implementation of a DEGAS object's variable method set. The implementation of a method is stored as a Python function. Each DEGAS object has a dictionary storing references to these Python functions indexed by method name, as discussed further in Section 7.1.2.

An additional attractive feature of Python is the extensive library [Rossum, 1995a] of modules available in the standard distribution. The modules in the library present their functionality as abstract data types. For example, the `socket` module implements Unix inter-process communication by a `socket` object with socket operations as methods. In the DEGAS prototype, we use several modules, that implement threads, locking, and inter-process communication. The `thread` module is used to implement a DEGAS object's separate thread of control. This module also implements simple locks to prevent conflicts between DEGAS objects and the system layer in the implementation of communication primitives. Finally, inter-process communication through sockets was used in the prototype implementation for communication between the DEGAS database and the user interface.

We also considered C++ [Stroustrup, 1991] as an implementation language. It satisfies our first two requirements, support for objects and for concurrency. It scores lower, however, on its fitness for prototyping. This is mainly due to the lack of higher level data structures in C++. Especially dictionaries and the various library modules of Python are easier to use than similar C++ facilities. These arguments also apply to Java. Furthermore, we did not have any portability requirements that could be fulfilled by Java.

7.1.2 The Basic DEGAS Object

This section discusses the implementation of the basic DEGAS object. It is implemented by a Python object. In this object, the data structures are attributes. The basic actions are method calls of the Python object implementing the DEGAS object.. The execution cycle in Figure 6.4 is also a method. To achieve concurrent execution of DEGAS objects, this method is executed in a separate thread for each object.

Storage

A basic DEGAS object records an object's capabilities, as specified in Section 6.2. Here, we look at the storage of these data in an implementation.

State History The state history of an object is stored as a dictionary with the time as a key. This allows fast retrieval of historical attribute values. Hence, the following data are recorded in a tuple:

1. `Time Stamp`. The time of the method call is represented by an integer.
2. `Attribute`. A dictionary containing name and values of the object's attribute set at the time indicated. The attribute name is the key of this dictionary.
3. `Method Name`. A string containing the name of the method call causing the state change.
4. `Method Parameters`. A list containing the parameters of the method call.

Queues The three queues of a DEGAS object are FIFO queues implemented by lists. These lists contains tuples representing the messages. The head of the list is the earliest message. The attributes of a message is dependent of the queue. The following attributes are used:

1. `Sender`. The object identity of the sender.
2. `ReplyTo`. The recipient of the message's response.
3. `Event`. This contains the encoded event expression of the selector. The encoding of event expressions is discussed in Section 7.1.2.
4. `Condition`. A selection condition is represented by a function testing it. This attribute of a query tuple contains a reference to this function.
5. `Method name`. A string containing the name of the method.
6. `Param`. A list of actual parameters of the method call.

The table below indicates the attributes of tuples in each queue of a DEGAS object.

	Sender	ReplyTo	Event	Condition	Method Name	Param
Query Queue	✓	✓	✓	✓		
External Method Queue	✓	✓			✓	✓
Internal Method Queue	✓	✓			✓	✓

The external queues, i.e., the query queue and the external method queue, are guarded by a lock, to avoid conflicts between the system layer and the object itself in modifying the external method queue. The list and the lock are encapsulated in a separate object.

`Reply Box` The `Reply Box` is a value.

Capability Sets The capability sets of a DEGAS object are stored in dictionaries. For each capability, attributes, methods, lifecycles, and rules, an object has a dictionary. Attributes and methods are stored indexed by their name, while rules and lifecycles are indexed by the name of their class or addon. This choices of indices is motivated by their most common use. Attributes and methods are always referenced by their names. Modifications of the lifecycle and rule dictionaries take place when the DEGAS object is extended by an addon or when an addon is removed. In these situations, the elements of the dictionary are accessed through the name of the defining addon.

As a result, the Python implementation of a DEGAS object has the following attributes:

1. **Attributes**. The (Python) attribute `attr` is the dictionary containing the object's current attributes and their values.

2. **Methods**. The attribute `meth` is the dictionary of methods. For every method of the DEGAS object, it contains a pointer to the Python function implementing the action.

3. **Lifecycles**. The attribute `cycl` is the dictionary containing the current lifecycle elements of the object. These are all lines from the lifecycle specifications in the class definition and the definitions of the addons currently present.

4. **Rule Set**. The attribute `rule` is the dictionary containing the event, the condition, and the action of all rules currently defined in the DEGAS object. The event is encoded as a finite state machine, as explained in Section 7.1.2. The condition is stored as a Boolean function checking it. The action is a method call.

`Lifecycle` The attribute `LifeCycle` is a dictionary containing the current lifecycle of the DEGAS object. It contains the finite state machine for the current lifecycle. The encoding of the state machine is explained in Section 7.1.2. It is constructed by combining the elements stored in the lifecycle dictionary `cycl` using the communication merge as explained in Section 5.7.2.

This The identity of an object is a logical address provided on creation by the DEGAS system layer. Hence, the attribute `This` in the DEGAS object is different from the self reference `self` of the Python object, which contains the physical address of the object.

Actions

The primitive actions of a basic DEGAS object are implemented by methods of the Python object. Here, we shortly give the techniques used by each primitive action.

ExecuteMethod A method is executed by looking up the reference to its Python translation in the `Meth` dictionary by its name. Then, the function containing the Python translation is executed. After that, the new object state is appended to the history. All constructs in a DEGAS method have a straightforward translation to Python code. The `Foreach ... in ... where ... do ...` can be implemented using the Python `For` iterator [Rossum, 1995b], that applies a program block to all elements of a set.

Below, we give the Python code implementing `ExecuteMethod`. The lines preceded by # indicate comments explaining the actions. First, the current attribute values are copied. After that, the actual execution takes place. The expression `self.meth[name]` resolves to the name of the Python implementation stored in the dictionary `meth`. The parameters taken by the Python function are the parameters of the DEGAS method and the current attributes. The presence of `self` as a parameter to the method call is a Python feature to indicate a function of a Python object. After execution of the method, the new attribute values are inserted in the object history.

```
def executeMethod(self,name,parameters):
    #
    # Copy attributes to local attribute dict
    #
    attribuut = {}
    lastattr = self.stateHist[max(self.stateHist.keys())]
    for i in lastattr.keys():
        attribuut[i] = lastattr[i]
    #
    # Execute the method
    #
    self.meth[name](self, parameters, attribuut)
    #
    # Append the local attribute dict to the state history
    #
    self.stateHist[self.count] = (attribuut,name)
```

CheckSelector The implementation of this action first checks the event expression using the techniques explained in Section 7.1.2. Then, satisfaction of the condition by the matching sub histories is checked. The condition is encoded as a Boolean function testing the condition.

MethodAllowed The implementation of this action takes the lifecycle stored as a finite automaton in `Lifecycle`. It checks the tail of the event history together with the proposed method call against the automaton. This is explained in Section 7.1.2.

get This is implemented by a generic `Get` method in the Python object, that returns the current value from the `Attr` dictionary.

Extend This action manipulates the capabilities sets, i.e., `Attr`, `Meth`, `Rules`, and `Cycl`, as specified in Section 6.2. A new automaton for `Lifecycle` is constructed using the compositions described in Section 7.1.2. A further explanation of the implementation of the addon mechanism is given below.

Remove This action is implemented analogously to the `Extend` action.

Kill Termination of an object means terminating its execution cycle. This is implemented by terminating the thread executing the object. The result is that the object is still around in the database, but that nothing is added to its history anymore. The object is available for historical queries.

Addon Extension

The implementation of the addon mechanism is supported by the storage of capabilities in dictionaries. Furthermore, the possibility of referencing functions by name allows an object access to functions that were not pre-defined in the object.

Recall the specification of the addon class object in Section 6.5. There, we saw that an object O executing an `Extend(`A`)` action requests the capabilities of addon A from the addon class object of A. References to functions simplify this process. The addon class object simply returns the name of a function containing the extension actions, i.e., the appropriate assignments to the capabilities sets. This function is executed by O to achieve the desired extension.

For example, suppose we have the specification of the addon `Extra` in Figure 7.1. For brevity, assume that the Python implementations of the methods represented by the functions `P_setFirst`, `P_setSecond`, and `P_total`, respectively. Furthermore, the encoding of the lifecycle is stored in `P_Lifecycle-Extra`. This results in the following function to extend a `Standard` object O with the addon `Extra`.

```
Addon Extra
Extends Standard
Attributes
   first : integer
   second : integer
Methods
   setFirst(no :integer) = {
      first := no
   }
   setSecond(no :integer) = {
      second := no
   }
   total = {
      return first + second
   }
Lifecycle
   setFirst*
   setSecond*
   total*
EndAddon
```

Figure 7.1: The specification of the Extra addon

```
def ExtendExtra(self) :
    Attr['first']       = 0
    Attr['second']      = 0
    Meth['setFirst']    = P_setFirst
    Meth['setSecond']   = P_setSecond
    Meth['total']       = P_total
    Cycl['Extra']       = P_LifecycleExtra
    sendMessage(ExtraClass,this,this,registerExtent)
```

The name of ExtendExtra is passed to *O* by the addon class object of Extra for execution. Removal of capabilities in a Remove action proceeds analogously to an Extend action. For the deletion of the addon from an object *O*, the addon class object of Extra passes the name of RemoveExtra to *O*:

```
def RemoveExtra(self) :
    del Attr['first']
    del Attr['second']
    del Meth['setFirst']
    del Meth['setSecond']
    del Meth['total']
    del Cycl['Extra']
    sendMessage(ExtraClass,this,this,removeExtent)
```

Checking Event Expressions

Event expressions occur in lifecycles and in selectors. In both cases, we must check (a part of) the event history against a process algebraic expression [Baeten and Weijland, 1990]. Here, we show that these checks are easy implementable. In fact, a finite automaton is sufficient to match the event history with an event expression. This also means that lifecycles in a DEGAS object have similar expression power as finite automata used to model the dynamic aspects of objects in OMT [Rumbaugh and others, 1991].

Proposition 1 *An event expression can be implemented by a finite state machine with conditions on the transitions.*

Proof Follows from the fact that the event expressions are a regular language [Lewis and Papadimitriou, 1981]. 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 is handled by rewriting them to the equivalent alternative composition. □

For each operator, we can give a simple automaton that checks this expression. These are shown in Figure 7.2. Please recall, that the negation operator $\neg$ can be rewritten to an alternative composition $+$. If an object O has action A, B, C, and D, then $\neg C = A + B + D$. Furthermore, the merge operation can be expanded to an alternative composition. The axioms of merge are as follows:

$$\begin{aligned} x \| y &= x \lfloor\!\lfloor y + y \lfloor\!\lfloor x \\ ax \lfloor\!\lfloor y &= a(x \| y) \\ a \| b &= ab + ba \\ (x \| y) \| z &= x \| (y \| z) \end{aligned}$$

As an example of a finite automaton to check a more complicated event expression, we give the automaton associated with the event expression $A; (B; \neg C; D)^*$ in Figure 7.3.

The implementation of these automata in Python is straightforward. Each state of the automaton is numbered. For each state, we record the possible transitions to other states. A transition is characterised by a state number and a method name. Consider the automaton in Figure 7.4, which implements the event expression $A; (B \| [\text{answer} = 42]C)$. In state 2 of this automaton, we have two possible transitions: To state 3 by method B and to state 4 by method C, if precondition answer = 42 is satisfied.

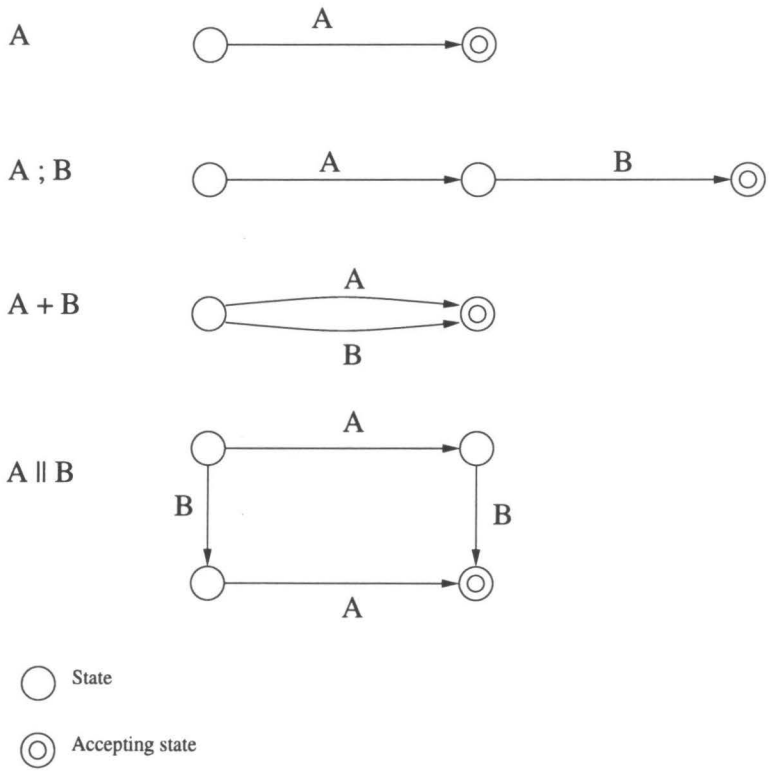


Figure 7.2: Finite automata for process algebraic operators

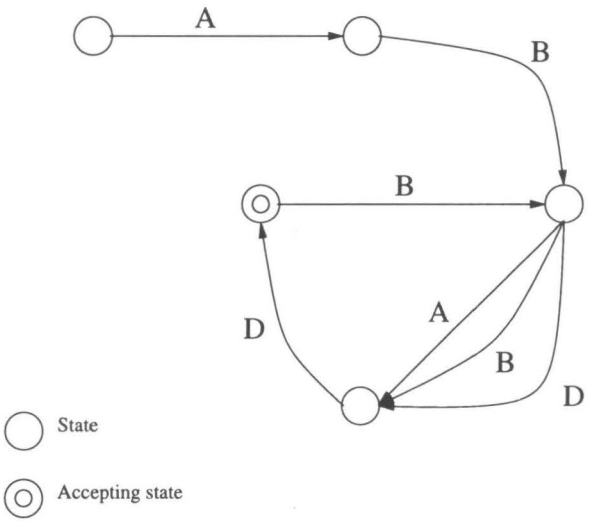


Figure 7.3: Finite automaton to check the event expression $A; (B; \neg C; D)^*$.

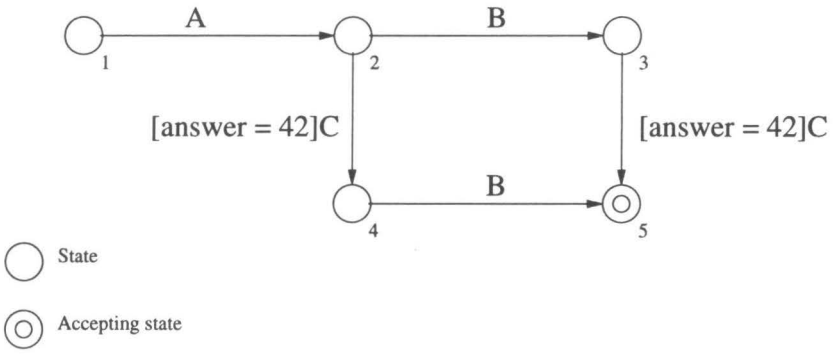


Figure 7.4: Finite automaton to check the event expression $A; (B \| [\text{answer} = 42]C)$.

Merging lifecycles from multiple specifications is a straightforward affair. In Section 5.7.2, we saw that lifecycles are composed using communication merge. Like simple parallel merge, communication merge is defined by the same axioms as parallel merge, with the addition of a communication function γ. This means that communication merge is associative, i.e., $(x|y)|z = x|(y|z)$. As a consequence, the sequence of lifecycle composition is arbitrary.

7.1.3 The System Layer

The underlying infrastructure of a DEGAS database is provided by the system layer specified in Section 6.3. Services provided are object creation and inter-object communication. The system layer is implemented by a Python object `DegasSystem`. Each site has an instance to implement the system layer. Its identity is known to every Python object in the implementation of the DEGAS database.

Attributes

The attributes of the `DegasSystem` object contain the data stored in the system layer, as specified in Section 6.3. This contains the directory of objects in two dictionaries. One maps DEGAS identities to Python identities. The other maps names to DEGAS identities.

`ObjDir` This is a dictionary mapping DEGAS object identities to Python object references. Object identities are represented by integers.

`NameDir` This is a dictionary mapping names of DEGAS objects to DEGAS object identities. Names are represented by strings, while DEGAS object identities are again represented by integers.

Object Creation

The creation of an empty DEGAS object is a straightforward process. We define a Python class `DegasObject`, that contains the capabilities of an empty DEGAS object, as specified in Section 6.2 and 7.1.2. The primitive `newObject` is a method of `DegasSystem` that yields a new instance of `DegasObject`. A DEGAS identity is assigned to the object, which is entered in a dictionary mapping DEGAS object identities to Python object references.

Inter-Object Communication

The DEGAS communication primitives were specified in Section 6.3. In particular, the effect of the primitives on the data structures of the destination object was specified. The implementation of these data structures, i.e., the query and method queues, was given in Section 7.1.2. The main task of the `DegasSystem`

object in communication actions, is to find the destination object. First, the destination must be resolved to a DEGAS object identity, if a name is given as destination. The name is resolved to an identity using `NameDir`. Second, the DEGAS identity must be resolved to a Python object reference in order to insert the message in the appropriate queue of the destination object. If the destination object is at the same site, the `DegasSystem` object will find its Python object reference in `ObjDir`. Otherwise, it needs to find out the location of the object in order to pass the message to the site of the object.

Object autonomy implies that a there is no centralised directory of all objects in the DEGAS database. Hence, a DEGAS database must implement a mechanism to find out the location of an object from the information at a site. A number of alternative schemes exist for this. A simple broadcast of a non-local message leads to a high load of the network. Alternatively, message for other sites can be posted on a "bulletin board", that is regularly checked by all sites. Although this reduces network traffic, it introduces a centralised resource, that forms a potential bottleneck in the system.

A problem analogous to locating objects also occurs in mobile computing [Imielinski and Badrinath, 1994]. Commonly, a mobile computing system is based on a cellular communication network. This raises the problem of determining in which cell a mobile computing node is. A number of schemes are proposed to solve this problem, e.g., in [Imielinski and Badrinath, 1992]. To examine the applicability of these to our problem, we translate these schemes to a DEGAS database.

An improvement on broadcasting proposed by [Imielinski and Badrinath, 1992] is to partition the network and record the partition an object is in. Thus, only a subset of the sites in the network needs to be consulted to find out the location of the destination object. Please note, however, that this requires the `DegasSystem` object to record object identities of all objects not present at its site. The same drawback applies to the two other approaches proposed. One approach is to list for each site the probability that an object is located there. To reach an object, these sites are tried by order of probability. The second approach creates a chain of pointers for each object. If an object leaves a site, the new location of the object is recorded. A message to an object follows these pointers. Although we can detect and remove cycles in such a chain, it still implies relatively long transit times for messages. It also leaves the problem, how a site determines the location of an object that never visited it. Moreover, the problem of locating objects occurred in the first place, because of the autonomy of objects not to inform the system of its whereabouts.

To stay in line with object autonomy, the task of tracing objects must be with the `DegasSystem` objects. It maintains a routing table by inspecting the message flow passing through. Recall from Section 4.5 that we assume for DEGAS a network based on links between sites. Furthermore, there is a `DegasSystem`

object at each site. If a message arrives at a site, it arrives over a specific link. Since each message contains its sender, the `DegasSystem` object can associate links with object identities in order to build a routing table. If an object identity is not found in the routing table, then the message is broadcast. This broadcast message is used to create the initial entry in the routing table[2]. If a DEGAS object moves to another site, the routing table might become outdated. This fact is identified if a site on the route does not know the destination object of the message. In this case, a new broadcast by the originating site is necessary.

7.1.4 Other Objects

All other required objects in a DEGAS database are DEGAS objects themselves. Their functionality is specified in the DEGAS programming language. Hence, the DEGAS object and the system layer are everything needed for the implementation of a DEGAS database.

7.2 Interface to the Outside

The specification in the previous chapter only discussed the components of the DEGAS system itself. It did not discuss the interfaces to the outside world, either human users or other systems. These interfaces are implemented by an object, whose interactions with DEGAS can be specified in DEGAS. In other words, it sends DEGAS messages to DEGAS objects and can receive DEGAS messages from DEGAS objects. Since this object must also communicate in another language than DEGAS, it will not be implemented in DEGAS itself.

To illustrate the interactions of DEGAS, we discuss two interactions with the environment, viz., data entry and queries.

Data Entry New objects are created by the action `New` in the class object, as specified in Section 6.4. The actual creation of objects and relation objects is identical. The creation of a DEGAS object is the way data is entered in a DEGAS database. This subsection shortly discusses the entry of data in a DEGAS database, i.e., its interface to the outside world. The creation of a relation object is part of establishing a relation, which is discussed in Section 7.3.

The requirement on interface objects are relatively loose. The object entering data, the *Data Entering Object* or DEO, only has to meet certain requirements

[2]The appropriate implementation of broadcast guarantees that the broadcast will yield the fastest route between two sites. This implementation means that a site forwards a broadcast message to all neighbours, except the one sending the message. The message identity is used to discard a message already seen. Obviously, a message reaches a site first over the fastest route. For an elaborate discussion of computer networks, the reader is referred to, e.g., [Tanenbaum, 1996].

regarding its interface with the DEGAS database. The other aspects of the DEO are indifferent to DEGAS, meaning that it can implement any kind of interface to a user, or to other software. If a DEO wishes to create an object of a class *ClassName*, it simply sends a `new` message to the class object. For example, the following message is sent to create a `Person` object:

NewOid := PersonClass.new()

The identity of the object is returned in `NewOid`, so that the DEO can enter data in the new object.

Query Interface Query requests are processed by the `Site` objects and the object instances, as specified in Sections 6.8 and 6.2. These requests are issued by a *query generating object* (or QGO). The only requirement on a QGO regards its interface to the DEGAS database. Apart from this, a QGO can implement any kind of query interface, be it a forms package, an SQL interface or a data mining client.

A query is distributed over all reachable sites in the networks by broadcasting a call to `distributeQuery` to all `Site` objects. To start processing a query $Q = \langle C, S \rangle$, where C is a class and S a selector, the QGO makes the following call:

Broadcast(Site, Self, Self, DistributeQuery(C,S))

To receive the answer to a query it issued, a QGO must implement the method `answerQuery`. This action receives the answer to a query from a `Site` object. As a parameter it takes the result of the site, which a set of object - history pairs.

$SiteResult : \mathcal{P}\langle Object : Oid, EH : EHist \rangle$

This method can be invoked by `Site` objects.

Prototype To further explain DEGAS' interface to the outside world, we shortly describe the user interface of our prototype. It is implemented as a separate process from the DEGAS database. The user interface is represented in the DEGAS database by the `DEGAS UI` object. The user interface sends text commands to the `DEGAS UI` object over a socket. These are then handed over as DEGAS messages to the system layer. Likewise, the answers to method calls or the results of queries are sent back by the `DEGAS UI` object as text. This setup is depicted in Figure 7.5.

7.3 Programming in DEGAS

The previous chapter specified requirements on the various parts of the DEGAS system. For relations, only a framework was specified, leaving the exact implementation to the application programmer. In order to give a further explanation of DEGAS relations, this section discusses programming in DEGAS.

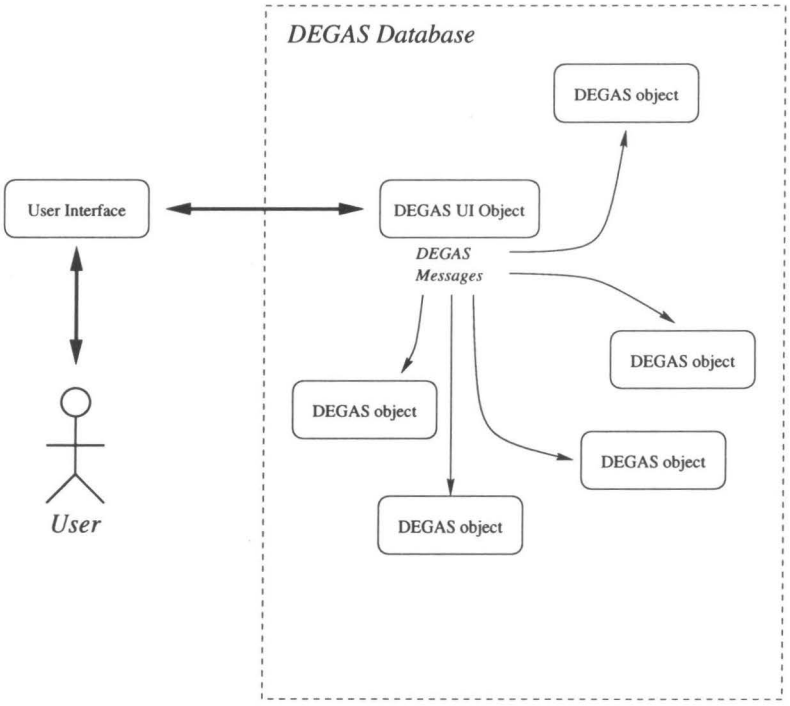


Figure 7.5: User interface of the DEGAS prototype.

The requirements DEGAS puts on the creation of a relation, are specified by the actions required in a relation class object in Section 6.7. As a consequence, the relation class object must collect the prospective partners and create the relation object. Other requirements on the creation of a relation are dependent on the semantics of the application. Hence, these are specified by the application programmer in the relation class object.

Initiative All applications have in common that the initiative for a relation is taken by an object. It indicates its intention to enter a relationship by sending an `initiate` message to the class object of the desired relation. The sender is recorded in the set `Prospects`. The relation is established, if the tuple in `Prospects` is completed. The relation class object tries to complete the tuple by obtaining `initiatePartner` messages from all prospective partners in a relation. The way `Prospects` tuples are completed is application dependent. The variations are:

1. One of the partners in the relation is fixed. An object expressing interest always engages in a relation with the same object.
2. An object expresses interest to engage in a relation with a specific partner object. The relation class object tries to establish a relation between the two specified object. The exact proceedings depends on whether prior agreement exists between prospective partners or not.
3. The relation class object matches partners. An object indicating interest in engaging in the relation is indifferent to its partner.

If one of the partners is fixed in the relation, the initiative is with the variable partner. The reaction of the relation class object to its `initiate` message is to send a message to the fixed partner. An example is found in the stock exchange scenario in Section 4.2. On receipt of an `initiateShareholder` message, the relation class object of the `Supply` relation will ask a known `MarketMaker` object to take the order. This is specified by the following rule:

```
  ⋮
Rules
   On initiateShareholder
   do myMarketMaker.takeBuyOrder(sender)
  ⋮
```

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. The specification of preconditions in a relation class object is discussed below.

The next variant is that one of the prospective partners expresses a desired partner. The difference with the previous case is, that the relation class object sends a request to that specific object. Suppose that we have a relation with partners `First` and `Second`, where the initiative is always with `First`. The method `initiateFirst` then becomes:

```
    ⋮
Methods
    initiateFirst(desiredPartner : oid) = {
        Prospects := Prospects + ⟨ First = sender, Second = Null ⟩
    }
    ⋮
```

The following rule sends a request to the desired partner:

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

The `initiateSecond` message confirms the relation:

```
    ⋮
Methods
    initiateSecond(partner : oid) = {
        Foreach p in Prospects
        where p.First = partner
        do {
            Prospects := Prospects - p
            instantiateRelation(p.First, sender)
        }
    }
    ⋮
```

The `instantiateRelation` message then creates the relation, as specified in Section 6.7.

The last kind of relation lacks all preferences. Hence, both partners can take the initiative for a relation. Furthermore, neither partner has a preference for a specific partner. In this case, the matching process is relatively simple. Since this case is symmetrical, we now name the roles `Left` and `Right`. We explain it for the `Left` partner only.

The `initiateLeft` method checks for the presence of unmatched `Right` prospects. These are collected in the set `RightProspects`.

```
    ⋮
  Methods
    initiateLeft() = {
      RightProspects = []
      Foreach p in Prospects
      where p.Left = Null and p.Right ≠ Null
      do RightProspects := RightProspects + p
    }
    ⋮
```

Then, the check on prospects of the other class is done by two rules:

```
    ⋮
  Rules
    On initiateLeft
      if RightProspects = ∅
      do enterProspectsLeft(sender)
    On initiateLeft
      if RightProspects ≠ ∅
      do pickRight(sender)
    ⋮
```

If there are no current prospective `Right` partners, then `EnterProspectsLeft` adds the sender of `initiateLeft` to `Prospects`. Otherwise, a partners is picked from `RightProspects` by the method `pickRight`. These two actions are specified as follows:

```
    ⋮
  Methods
    enterProspectsLeft(prospectLeft : oid) = {
      Prospects := Prospects + ⟨ Left = prospectLeft, Right = Null ⟩
    }
    pickRight(prospectLeft : oid) = {
      p := head(RightProspects)
      p.Left := prospectLeft
      instantiateRelation(prospectLeft, p.Right)
    }
    ⋮
```

Preconditions Besides the agreement of the partners, a relation can have other preconditions. These are checked by the relation class object. If the preconditions are satisfied, the relation can be created by the `instantiateRelation` method.

To specify the preconditions of a relation, we can use the DEGAS lifecycle mechanism. In the lifecycle of a relation class object, the relation's preconditions

would be the guard conditions to the action that creates the relation object. This specification of a relation's preconditions, however, 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
   ⋮
```

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 application programmer. This specification, however, is located in the relation object instead of the relation class object. Usually, this proceeds analogous to the creation: One of the partners indicates that it wishes to end the relation. Dependent on the relation, the other partner is asked whether he agrees. Possible conditions on the termination of a relation can be checked in the lifecycle of the relation object.

Example To conclude this discussion, we return to the stock exchange example from Section 4.2. Figure 7.6 gives the complete DEGAS specification of the `Supply` relation class object from the stock exchange example. The attribute `SupplySet` contains all current instances of the `Supply` relation.

The script in Figure 7.7 shows the messages exchanged in order to create a `Supply` relation between a `Shareholder` S and a `MarketMaker` M via the `Supply` class object SC.

7.4 Query Processing in DEGAS

Chapter 6 specified the actions of various objects that facilitate query processing. In this section, we explain how these actions are related by showing the

```
Object SupplyClass
Attributes
   Extent : POid
   Prospects : P⟨ Shareholder : Oid, MarketMaker : Oid ⟩
   myMarketMaker : oid
   RelationId : Oid
Methods
   initiateShareholder = {
      Prospects := Prospects + ⟨ Shareholder = Sender, MarketMaker = Null ⟩
   }
   initiateMarketMaker(Shareholder : Oid ) = {
      Foreach p in Prospects
      where p.Shareholder = Shareholder
      do {
         p.MarketMaker := Sender
         instantiateRelation(Shareholder, Sender)
      }
   }
   instantiateRelation(Shareholder : oid, MarketMaker : oid) = {
      Shareholder.extend(ForSale)
      MarketMaker.extend(Supplied)
      RelationId := new()
      Shareholder.initialiseForSale(RelationId)
      MarketMaker.initialiseSupplied(RelationId)
      Foreach p in Prospects
      where p.Shareholder = Shareholder and p.MarketMaker = MarketMaker
      do Prospects := Prospects - p
   }
   cancelInitialisation(Shareholder:oid) = {
      Foreach p in Prospects
      where p.Shareholder = Shareholder
      do {
         Prospects := Prospects - p
         Shareholder.cancelSupply
   }
Lifecycles
   (initiateShareholder;initiateMarketMaker;
      (instantiateRelation+cancelInitialisation))*
Rules
   On initiateShareholder
      do marketMaker.takeBuyOrder(Sender)
   On initiateShareholder(Time);Time + 30 min
      do cancelInitialisation(Sender)
EndObject
```

Figure 7.6: DEGAS specification of the `Supply` relation class object.

S	*M*	*SC*
SC.initiateShareholder		
		M.takeBuyOrder
	SC.initiateMarketMaker	
		instantiateRelation(*S*,*M*)
		S.extend(ForSale)
		M.extend(Supplied)
		RelId = new()
		S.initialiseForSale(*RelId*)
		M.initialiseSupplied(*RelId*)

Figure 7.7: Script to create a `Supply` relation

execution of a query. Object autonomy is a complicating factor in query processing. DEGAS does not honour a number of assumptions usually made in distributed query process, as stated by [Stonebraker *et al.*, 1996]:

- Exact knowledge of the data fragmentation of the database.
- A fixed allocation of the data in the database.
- Uniformity of the nodes and their connections.

In particular, DEGAS' object autonomy not only means that objects are free to move between sites, but also, and more importantly, that they need not necessarily answer every query received.

In this section, we discuss the resulting method for processing a query in DEGAS. Furthermore, we discuss how we can derive the information needed to maintain a data dictionary using data from query results. Finally, we discuss the approximation of a query result's quality.

7.4.1 Processing a Query

As a consequence of object autonomy, there is no central directory of objects in a class. This means that a query cannot be sent directly to the objects. Instead, it is sent to the sites for local distribution. It is up to the DEGAS objects to decide whether to answer the query or not.

Recall from Section 5.9, that a query Q is a pair $\langle C, S \rangle$ of a class C and a selector S. A query $Q = \langle C, S \rangle$ is issued by a QGO, which was discussed in Section 7.2. The QGO broadcasts the query to all `Site` objects. Each `Site` object checks the existence of the attributes and methods in the selector using the `isAttribute` and `isMethod` methods of the class object/ Then, it collects the local extent $\mathcal{L}$ of C and sends the query to all elements of $\mathcal{L}$. These return their results to

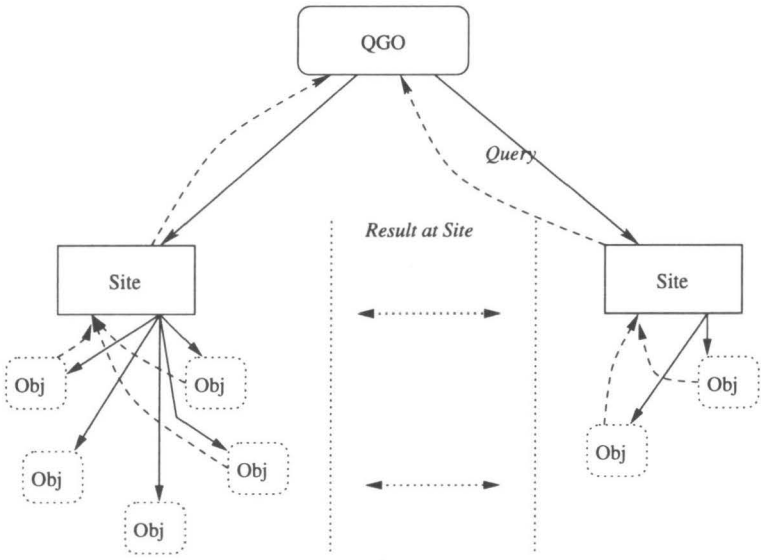


Figure 7.8: The information flow in DEGAS query processing

the `Site` object, which collects them for shipment to the QGO. This flow of information is depicted by Figure 7.8.

As an illustration, consider the following query selecting bank accounts that were overdrawn by a single large transaction:

Select from BankAccount
on debit(db_amount)(t)[1 Mar 1997, 15 Mar 1997]
if db_amount > 10000 **and** balance(t) ≤ 0

The QGO for this query makes the following call to send the query to the sites:

```
Broadcast(Site,
        DistributeQuery(BankAccount,
                        ⟨debit(db_amount)(t)[1 Mar 1997, 15 Mar 1997],
                         db_amount > 10000 and balance(t) ≤ 0 ⟩
                        )
        )
```

Then, each `Site` object executes the method `distributeQuery`, which, after checking the type correctness, assigns an identity qid to the query and then forwards it to each `BankAccount` object at the site. Each object executes a `checkSelector` for the selector consisting of event

debit(db_amount)(t)[1 Mar 1997, 15 Mar 1997]

and condition

db_amount > 10000 **and** balance(t) ≤ 0 ⟩

The result is a set of matching sub-histories. In this case, these are occurrences of the `debit` event satisfying the condition. These are collected in a result R, which is sent back to the `Site` object by the following call:

Site.queryResult(qid,R)

The `Site` object collects the results from the individual objects in a local result LR. The complete result is sent back to the QGO by executing the method `shipResult` of the `Site` object, which calls the method `answerQuery` in the QGO. The call to `shipResult` is commonly triggered by a rule monitoring the time elapsed after execution of `distributeQuery`. To return the local result $\mathcal{LR}$ to the QGO, the following call is made by `shipResult`:

QGO.answerQuery($\mathcal{LR}$)

The table in Figure 7.9 summarises the calls made in query processing

QGO	Site S	Class Object C	Instances I
S.distributeQuery			
	C.isAttribute(...)		
		return *true* to S	
	C.isMethod(...)		
		return *true* to S	
	C.getExtent		
		return $\mathcal{L}$ to S	
	$\forall I \in \mathcal{L} : I$.sendQuery		
			checkSelector
			S.queryResult
	shipResult		
	QGO.answerQuery		

Figure 7.9: Calls made in DEGAS query processing

7.4.2 Maintaining a Data Dictionary

A practical element of a database management system is the data dictionary. Object autonomy in DEGAS has a significant impact on the maintenance of a data dictionary. First, the distributed nature of DEGAS implies a distributed data dictionary. Second, object autonomy means that the data dictionary must actively collect the required (meta)data. A natural place to maintain a data dictionary is the site. Hence, a `Site` object is responsible for maintaining the data dictionary.

The contents of a data dictionary consists of schema information and fragmentation data. In Section 6.4, we saw that schema information is maintained in class objects. At each site, if an object of class C is present on the site, then a class object for class C must be present on the site. The reverse need not be true. Hence, we maintain schema information about at least the local schema. Since class objects have been specified in Section 6.4, we do not discuss this further in this section.

The fragmentation data in the data dictionary contains information on the number of objects on other sites. For each class in the local schema, i.e., for each class it knows, it stores the number of objects in the class extent at each known site. Hence, the data dictionary is a table with sites as rows and classes as columns. An example is the following table:

	Employee	Customer	Article	...
db_1	123	343	456	...
db_3	22	238	309	...
db_4	32	394	234	...
db_7	56	387	105	...
⋮	⋮	⋮	⋮	

Since there is no directory of connected sites in a DEGAS database, the only means to enquire for dictionary information is by a broadcast. Due to network failures, periodical broadcasts by each site are no guarantee for the completeness of the information it acquires. Hence, for reasons of efficiency, we piggyback the information exchange to the query results returned by the sites. The query results can be accompanied with information about itself, which is then used to fill the data dictionary. Hence, no overhead is added and the frequency of updates to the data dictionary reflects the intensity of use. The piggybacking strategy also ensures that frequent queries on a site mean frequent updates to the data dictionary.

In the piggybacking strategy, the following information is sent with a query result:

- Site identity.
- Time the query arrived at site.
- Number of relevant objects at site.
- Number of relevant objects at site that answered.
- Number of objects in the query result

This information is then passed to the site of the QGO, i.e., the originating site of the query, to update the data dictionary.

To show the effect of a query result on the data dictionary, we give an example. Suppose that at time t_1 we have the left hand entry given in Figure 7.10 for the `Employee` class in the data dictionary.

site	#Employee
s1	23
s2	145
s4	12
s6	234
s10	34
s12	9
s23	322

site	#Employee
s1	21
s2	55
s4	17
s6	34
s10	35
s12	9
s23	332

Figure 7.10: Data dictionary entry for class `Employee` at time t_1 (left) and t_2 (right)

The information in this histogram is based on the results returned by the sites in the left hand column. At a later time t_2 we get an answer to a query for `Employee` objects from sites $s1, s2, s4, s10$ and $s23$. The data dictionary is updated to contain the data, resulting in the right hand entry in Figure 7.10

Now, the data on $s6$ and $s12$ dates from t_1, while the rest of the data dates from t_2. Thus, the histogram represents different time points for different sites. In general, however, the entries of most sites will be within a reasonable range from each other. Furthermore, at a query-intensive site, queries will be issued often enough to ensure small time differences between sites, unless a site is off-line for a long time.

7.4.3 Approximating Quality

In Section 4.4, we briefly touched on the notion of query quality. Here, we discuss a method to approximate the quality of a query result. For this approximation, we use the information in the data dictionary and the update information returned with the queries. Intuitively, the quality of a query result is the number of objects that is retrieved relative to the number of objects satisfying the query. Hence, calculating the quality of a query result means estimating:

$$Q = \frac{R^+}{C^+}$$

where R^+ denotes the number of objects returned by the query as satisfying the selector and C^+ denotes the actual number of objects in the database satisfying the selector. In this fraction, C^+ is the number to be estimated.

In DEGAS query processing, there are two sources of uncertainty, that may lead to missing objects satisfying a query. The first source is found at the site level. Due to object autonomy, a number of objects may not answer the query. The second source of uncertainty is the state of the network. If the network is partitioned, some sites, and the objects at those sites, may be unreachable. These two sources are reflected in the way we estimate the quality of a query. First, the quality of the query result is calculated for each answering site. Then, this data together with the data from the data dictionary of the originating site is used to estimate the total quality of the query result.

For each site, we can split up the objects in the local class extent into the following three categories:

A^+	#objects that answer and satisfy the selector
A^-	#objects that answer and do not satisfy the selector
NA^+	#objects that do not answer and satisfy the selector
NA^-	#objects that do not answer and do not satisfy the selector

This also gives us $A = A^+ + A^-$, the total number of objects that answer, and $NA = NA^+ + NA^-$, the total number of objects that do not answer. In addition, we denote the number of objects in the local class extent by S. In analogy to A, we denote by S^+ and S^- the number of objects in the local class extent respectively satisfying and not satisfying the query selector.

The quality of the result at the site now is:

$$Q_S = \frac{A^+}{S^+} = \frac{A^+}{A^+ + NA^+}$$

Since $S^+ = A^+ + NA^+$, we have to estimate NA^+. To estimate this number, we assume that answering a query is independent of satisfying the query selector. In practical terms, this means that the proportion of the non-answering objects satisfying the query selector is equal to the same proportion for answering objects. This proportion is given by A^+/A. The estimation for NA^+ is:

$$NA^+ = \frac{A^+}{A} \cdot NA$$

The quality Q_S then becomes:

$$\begin{aligned} Q_S &= \frac{A^+}{A^+ + \frac{A^+}{A} \cdot NA} = \frac{A^+}{A^+ + \frac{A^+}{A} \cdot (S - A)} = \frac{A^+}{\frac{A^+ \cdot S}{A}} \\ &= \frac{A}{S} \end{aligned}$$

The next step in the estimation of query quality is the combination of the per-site qualities, taking the unreachable sites into account. Here, we use the following numbers:

NS	#objects at non-answering sites
NS^+	#objects at non-answering sites that satisfy query selector
S_{DD}	#objects at site according to the data dictionary

The quality for the complete query is calculated by taking the weighted average of the per-site qualities. Since we are interested in the set of objects satisfying the query, we use the number of objects satisfying the query as the weights. This means S^+ for reachable (i.e., answering) sites and NS^+ for non-reachable sites. NS^+ is unknown, which means that we have to estimate it. To make this estimation, we assume that the proportion of objects satisfying the query at non-answering sites is equal to the average of same proportion at answering sites.

$$\forall i \in \rho : P_i = \frac{S^+}{S}$$

Then:

$$\frac{NS^+}{NS} = \bar{P}$$

where $\bar{P}$ denotes the average of all P_i. To give an estimate of NS^+ for each unreachable site, we use the data dictionary. There, we can look up the latest number we have for S, which is denoted by S_{DD}. Hence, the estimate for NS^+ becomes:

$$NS^+ = \bar{P} \cdot S_{DD}$$

We denote the set of reachable sites by ρ and the set of non-reachable sites by $\neg\rho$. Since we do not obtain any objects from unreachable sites, the result quality of these sites equals zero. Hence, the total quality Q becomes:

$$\begin{aligned}
Q &= \frac{\sum_{i\in\rho} S_i^+ \cdot Q_i + \sum_{j\in\neg\rho} NS_i^+ \cdot Q_j}{\sum_{i\in\rho} S_i^+ + \sum_{j\in\neg\rho} NS_i^+} \\
&= \frac{\sum_{i\in\rho} S_i^+ \cdot \frac{A_i^+}{S_i^+}}{\sum_{i\in\rho} S_i^+ + \sum_{j\in\neg\rho} NS_i^+} \\
&= \frac{\sum_{i\in\rho} A_i^+}{\sum_{i\in\rho} S_i^+ + \sum_{j\in\neg\rho} \bar{P} \cdot S_{j,DD}} \\
&= \frac{\sum_{i\in\rho} A_i^+}{\sum_{i\in\rho} S_i^+ + \bar{P} \cdot \sum_{j\in\neg\rho} S_{j,DD}}
\end{aligned}$$

This nicely corresponds to the notion that the quality of a query result is the proportion of objects returned from the total number of objects satisfying the query.

$$Q = \frac{R^+}{C^+}$$

where R^+ is the total number of objects returned and C^+ is the total number of objects in the root class of the query satisfying the query selector.

$$R^+ = \sum_{i \in \rho} A_i^+$$

$$C^+ = \sum_{i \in \rho} S_i^+ + \sum_{j \in \neg\rho} NS_j^+$$

Example Suppose we have the following simple query in a DEGAS database, where the data dictionary entry for `Employee` is the one given in Figure 7.10.

Select from Employee
where salary > 150000

The data dictionary entry for class `Employee` at the originating site of query is as follows:

Class Employee	
site	number
s1	21
s2	55
s4	17
s6	34
s10	35
s12	9
s23	332

We get an answer from all sites, except $s4$ and $s10$. The following number of objects are reported in the query result:

Site i	S_i	A_i	A_i^+
s1	25	20	4
s2	155	132	34
s6	243	243	49
s12	9	9	2
s23	342	298	53

Now, we can give the following estimates for P_i and S_i^+:

Site	P_i	S_i^+
s1	0.200	5
s2	0.258	40
s6	0.202	49
s12	0.222	2
s23	0.178	61

Further, we can calculate that $\bar{P} = 0.212$. The entries for $s4$ and $s10$ in the data dictionary give us $S_{s4,DD} = 17$ and $S_{s10,DD} = 35$. Hence, the quality of this result is estimated to be:

$$Q = \frac{4 + 34 + 49 + 2 + 53}{(5 + 40 + 49 + 2 + 61) + (0.212 \cdot (17 + 35))} = \frac{142}{157 + 11} = 0.845$$

More complex cases In general, a number of classes can be involved in a query. There are two ways to involve more than one class in a query. The first is to include path expressions to other object classes in a query. The second is to combine two classes using a nested query.

The quality of the result of a path expression is measured through the root class of the path expression. If one of the objects on the path does not answer, the root object will not return a value for the path expression. Hence, the quality of a path expression is the proportion of objects of the root class that answers. Formally, if we have a path expression

$$c.a_1.a_2.a_3.a_4$$

then the quality is given by:

$$\frac{Q_c}{Q_c^*}$$

where Q_c is the number of answering objects of class c and Q_c^* the estimated number of objects in class c.

In the case of a nested query, we do not calculate a composite quality measure. The inner query is a complete query, so a user can separately specify a desired quality for it. The specification of a combined quality would imply less control for the user than desirable. For example, if we combined the quality by multiplying the qualities of the inner and the outer query, then we would equate a quality of 25% times 80% with a quality of 40% times 50%. The latter may be more acceptable, since it is based on a more balanced sample of the database.

7.5 Conclusion

This chapter discussed a number of practical issues in constructing a DEGAS database. The first was the implementation of DEGAS. We motivated the choice of Python as an implementation language, based on its object-oriented features and its suitability for prototyping. This facilitates easy implementation of a basic DEGAS object. The variable sets of capabilities in a DEGAS object are implemented using dictionaries storing the capabilities by name. The thread of execution of an object is provided by the Python threads library.

The implementation of local communication on a site is straightforward using a simple object directory. Objects at other sites are located, taking object autonomy into account, through routing tables based on the observations of the

`DegasSystem` objects. Other required objects in a DEGAS database are DEGAS objects themselves. Hence, the feasibility of implementing DEGAS object implies the feasibility of their implementation.

A further issue addressed in this chapter concerned the interaction of DEGAS with users or other systems. These are represented by objects, whose interactions with DEGAS are specified in DEGAS. An example discussed was the Query Generating Object. Furthermore, this chapter discussed programming in DEGAS. In particular, we discussed how different application semantics of initiating relations are programmed. The functional specification in Chapter 6 defined a framework, that can be filled in in a number of ways. We showed how rules and lifecycles are used for this purpose.

Finally, we discussed query processing in DEGAS. Object autonomy has its consequences on the way we calculate the result of a query. In particular, it leads to an approach based on broadcasting. Furthermore, the maintenance of a data dictionary is based on information piggybacked on query responses. Since object autonomy gives an object the freedom not to answer a query, the proportion of objects responding becomes important. This is captured in the notion of the quality of a query's result, which is approximated using data returned with the query and data recorded in the data dictionary.

Part III

Design

Chapter 8

Modelling Workflow in DEGAS

In Section 3.1.1, we pointed out the use of active rules to encode large parts of an information system's dynamics. Hence, the specification of rules becomes an integral part of information system and database design. In this chapter, we discuss the design of a DEGAS database. Here, the most important aspect is the modularisation of data and behaviour. Modularisation is generally accepted as a necessary tool for the design and understanding of computer software. Naturally, this also applies to rules in active databases. As we discussed in Section 3.4.1, there are two approaches to the modularisation of rules. Either specific modularisation mechanisms are applied to the rules orthogonal to other modularisation, or one integral modularisation mechanism is applied to all elements of the database.

Research on active database design has mainly focussed on analysing rule sets in order to check desirable properties, such as termination and confluence. In this area, modularisation has also been addressed, e.g., in [Baralis *et al.*, 1996]. This work, however, focussed on partitioning a given rule set. Relatively little attention has been paid to the question, how we get a rule set for a certain application in the first place. In this chapter, we investigate this issue. In particular, we look at the elements of an object-oriented design method, such as OMT [Rumbaugh and others, 1991], that are of importance to the derivation of rules in an application.

As an example in this chapter, we use workflow. A workflow is an activity involving the coordinated execution of multiple tasks performed by different processing entities [Rusinkiewicz and Sheth, 1995]. The work of many organisations is centered around workflows. Classical examples of workflow are the processing of insurance claims and processing of loan requests. Current implementations of workflow are mostly in separate workflow management systems (WFMS). Since most workflow management involves large amounts of data, heavy interaction takes place between the WFMS and the database. At the same time, the Event-Condition-Action rules of active databases add to a database

management system the kind of reactive capabilities also found in a WFMS. Hence, we expect the integration of workflow management into active database management systems to be beneficial.

Previous work on workflows in active databases is reported in [Casati *et al.*, 1996a] and [Jasper *et al.*, 1995]. In general, the use of an active database improves the data handling capabilities in the workflow. This pertains to the application data, as well as to the workflow management data. Hence, approaches to workflow based on active databases such as [Casati *et al.*, 1996a] provide models of the data involved in the workflow. This is in contrast with work such as reported in [Aalst *et al.*, 1994], based on Petri nets, with an inherent focus on the specification of the dynamics of a workflow.

A drawback of the approach in [Casati *et al.*, 1996a] is the lack of modularisation in the rulebase. A large set of rules is generated for a workflow, which is only partitioned afterwards for analysis purposes. Hence, a separate modularisation is applied to the rules. Furthermore, this modularisation is not used in the design phase. In this chapter, we consider the design of a workflow with the other of the two approaches to rule modularisation mentioned above. In addition, we make the modularisation during the design of the application, using design principles formulated in this chapter. We show that DEGAS allows us to modularise workflow management in a way that separates concerns and that promotes flexibility. In particular, it offers a framework to implement the workflow evolution policies described in [Casati *et al.*, 1996b].

We state the DEGAS database design principles in Section 8.1. The next section, Section 8.2, discusses the specification of a workflow. In Section 8.3, we apply the guidelines to develop a design for workflow enactment. The following section, Section 8.4, shows that this design offers the necessary flexibility for evolution of the workflow. As usual, the last section contains conclusions.

8.1 Design Guidelines for DEGAS

In modelling applications for DEGAS, we need a number of guidelines. Since object autonomy is a central notion in DEGAS, we first recapitulate the criteria for object autonomy given in Section 2.4:

- Every object has a separate thread of execution.
- Complete encapsulation of an object's behaviour.
- Strictly regulated access to an object.
- Minimal guarantees about an object's behaviour towards other objects.
- Minimal dependency of an object on the behaviour of other object.

- Autonomy must be given up explicitly.

These criteria were used to guide the development of the DEGAS data model, introduced in Chapter 4. Naturally, they have their consequences on DEGAS database design. In one sentence, we can say that DEGAS objects combine minimal capabilities with maximal encapsulation.

Minimal capabilities means, that at any time an object only possesses the capabilities it needs. This applies to both time and place of information storage. An object gets information only when it needs it. Likewise, if it needs information from another object, it will request that information when needed. An application of this principle is, that an object is extended with extra capabilities, when it enters a relation, as explained in Section 4.1. If it is not in a relation, the object does not have the information associated with that relation.

Maximal encapsulation means, that everything is defined on the object itself. It is one of the main consequences of object autonomy, introduced in Section 4.1. Every aspect of the behaviour of an object is defined on the object itself, including rules. The modularisation primitives for the rules are the notions of object-orientation, that are also applied to attributes, methods, and lifecycles.

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

These design principles are applied to a database design process. Design in DEGAS encompasses four dependent phases, that follow from the architecture of the DEGAS model. In particular, each phase focuses on one category of DEGAS object capabilities, i.e., attributes, methods, rules, and lifecycles.

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

Each phase in DEGAS database design is executed iteratively. First, the inherent capabilities of objects are addressed. Then, we specify the capabilities of addons. This iteration originates in the semantics of DEGAS addons. In the specification of an addon, we can use those capabilities of the object being extended, that we are certain to be present. In other words, an addon can use the capabilities inherent to the object it extends, and the addons it assumes present, as declared in the `extends` specification. If we assume a directed edge from

each relation to its partner objects, then a DEGAS database design is a Directed Acyclic Graph (DAG) with the objects as the leaves, as is depicted in Figure 8.1. Hence, we can start out from the leaves in the DAG and then progress towards the highest level relations.

Phase 1 The first phase of database design in DEGAS is concerned with the static part. It consists of the identification of the objects and their relations, and the determination of the information contained in them. The identification of objects and relation objects in a DEGAS database design is not radically different from usual object-oriented design techniques [Rumbaugh and others, 1991]. Next, we determine the data in each object. Here, we have to make a distinction between the data that is always present, and the data that is dependent on the presence of a relation with another object.

As was discussed in Section 3.4.4, the part of an object associated with a certain relation is generally called its *role* in the relation. The concept of roles in an object-oriented context is elaborately discussed in [Wieringa *et al.*, 1995]. The capabilities an object has to deal with a certain relation are modelled by a role. Since these capabilities are only needed when the object is involved in the relation, these are called *transient* in contrast with the object's permanently present inherent capabilities. In DEGAS, transient capabilities of objects are defined in addons. Hence, all information associated with a role is implemented by an addon.

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

The result of this design phase is a static DEGAS object model. We have defined the objects, their relation and associated addons. Furthermore, we have defined the attributes in each of these.

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

Initially, the approach to finding the methods is a "shopping list" approach, as it is called by [Meyer, 1988]. The actions of an object can be either services to the outside world, or internal state transitions. In phase 2, this distinction is not of importance.

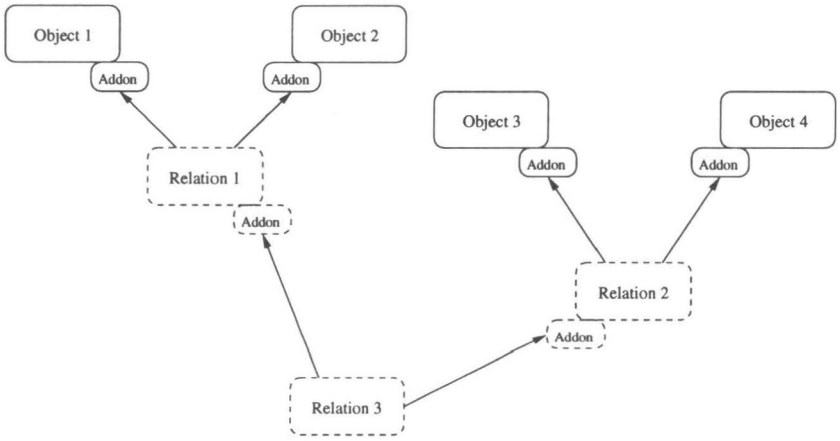


Figure 8.1: A DEGAS database design as a Directed Acyclic Graph

This phase gives us information to check the result of the previous phase. For each of the actions, we can determine the information it needs. This information must be either available in the object itself, in the form of an attribute, or it is obtained through a relation. Hence, we can check whether we have specified all attributes of an object. Furthermore, there must be a relation for every information exchange between objects. Vice versa, we can check whether every relation is used to exchange information between objects.

Phase 3 Having specified the actions an object can execute, we have to specify when these actions are executed. We do this by specifying the situations that trigger the actions. The specification formalism of these situations is up to the designer. He can use a graphical formalism, such as the situation diagrams introduced in [Lang *et al.*, 1996], or another formalism. The only requirement is, that it has a clear translation to ECA rules.

In this phase, we first specify the activation conditions inherent in an object, i.e., an object without any addons. After that, we specify the interaction scenario for each relation in an appropriate formalism, e.g., the event trace and event flow diagrams of OMT [Rumbaugh and others, 1991]. The interaction scenario describes the communication between the partners in the relation and the relation object. From this scenario, we derive the activation conditions for the actions of the objects. Please note, that these interaction scenarios can involve inherent actions of a partner object, since these are available to addons.

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

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

The result of this phase is the specification of an activation condition for each object's actions. These activation conditions are translated to rules. Furthermore, we can validate the list of actions specified in the previous phase using the interaction scenarios. If we are not able to formulate an activation condition for an action, it is most probably not needed in the current application.

Phase 4 The specification of the temporal ordering of actions forms the last phase in the design of a DEGAS database. These are meant to express the ways an object can execute actions. The result of this phase is the lifecycle of an object. The lifecycle specified for an addon conforms to the lifecycle of the inherent object by the use of the communication merge to merge the lifecycles of objects and addons, as discussed in Section 5.7.2. In terms of OMT, the lifecycle gives the dynamic model of the application, without the activations of the state transitions.

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

8.2 Specification of workflow

The example to show the DEGAS design process is workflow management. A workflow is a coordinated activity of multiple processing agents. Each agent executes a task, that is a part of the activity. Since the activity is usually embodied by an information object passed from one agent to another, e.g., a form in a paper-based workflow, the order of task execution is called the routing of an activity. In the specification of workflow, routing of an activity is our main concern. This specification is called the *schema* of the workflow. For each processing phase, we store the immediately preceding and succeeding tasks. This specification of workflow routing allows us to specify the conditions to start each processing phase in the workflow. These conditions depend on the way the phase is related to its predecessors. The different types of relations between tasks are called *routing elements.*

The set of preceding tasks of a task τ is denoted by the set $pred(\tau)$. Likewise, the set of succeeding tasks is denoted by $succ(\tau)$. As an example consider the workflow in Figure 8.2, which gives the workflow for billing an order. A bill is

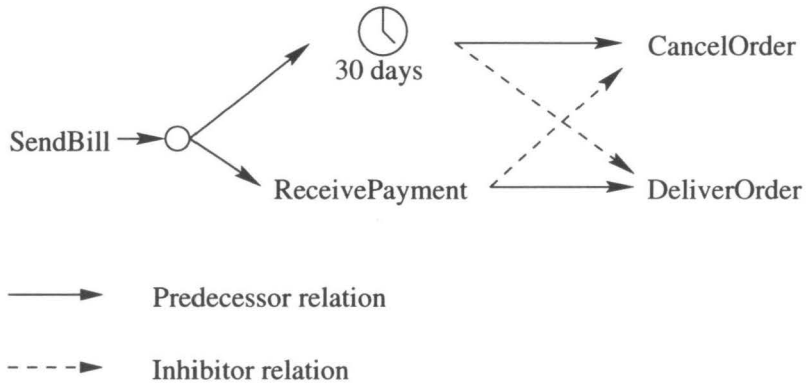


Figure 8.2: Part of an order processing workflow with a clock and task inhibition

sent to the customer. If we do not receive payment in 30 days, the order is cancelled. Otherwise, the order is delivered.

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

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

$$pred(\text{sendBill}) = \emptyset$$
$$pred(\text{timer(30)}) = \{\text{sendBill}\}$$
$$pred(\text{ReceivePayment}) = \{\text{sendBill}\}$$
$$pred(\text{CancelOrder}) = \{\text{timer(30)}\}$$
$$pred(\text{DeliverOrder}) = \{\text{ReceivePayment}\}$$

Besides positive predecessors, we need the notion of negative predecessors, or *inhibitors* for a task. These are tasks, that prevent the execution of another task. In the above example, we cancel an order of a customer who has not paid his bill in 30 days. Clearly, this task is inhibited by the payment of the bill. To specify this, each task τ has a set of inhibiting tasks denoted by $Inhib(\tau)$. In the example in Figure 8.2 we have:

$$Inhib(\text{CancelOrder}) = \{\text{ReceivePayment}\}$$
$$Inhib(\text{DeliverOrder}) = \{\text{timer(30)}\}$$

Other tasks have an empty set of inhibiting tasks. A task cannot be executed, if one of its inhibiting task is finished before its start.

The example in Figure 8.2 also contains a special kind of task, viz., a timer. A timer is simply a task that is completed at the specified time past its start.

In this example, the effect is that the ReceivePayment task only inhibits the cancellation of an order, if it is completed before 30 days are over.

Inhibitors are not the only means to prevent the execution of a task. There might be certain conditions associated with the execution of a task on a job. One of the main uses of conditions is as a criterion to choose between a number of successor. For example, the billing procedure of a mail order company might make a difference between new customers and known customers. Hence, each task τ has an associated precondition $Precond(\tau)$. If τ has no specific precondition, $precond(\tau) = True$.

The formalisation given above can specify all possible routings in workflow management. All possible routings can be composed from a finite number of routing elements. The Workflow Management Coalition[1] [Workflow Management Coalition, 1996] distinguishes five routing elements, apart from simple sequential routing. These are AND-split, AND-join, OR-split, OR-join, and iteration. We show that these routing elements can be formalised in terms of predecessor and successor tasks, and preconditions. Hence, our formalisation can specify all routings composed from these routing elements. Furthermore, we give a translation in terms of active rules to start each succeeding task.

Theorem 5 *All routing elements defined by the Workflow Management Coalition can be specified by* DEGAS *rules.*

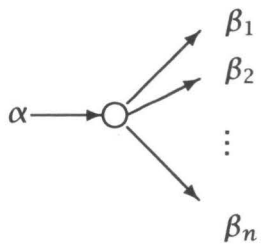
Proof The Workflow Management Coalition identifies five different routing elements:

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

A split means that a task has multiple successors, while a join means multiple predecessors. AND means that all successors or predecessors are involved. Likewise, OR means that only one of the successors or predecessors is involved. We show that all these elements can be specified by DEGAS rules by giving these rules.

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

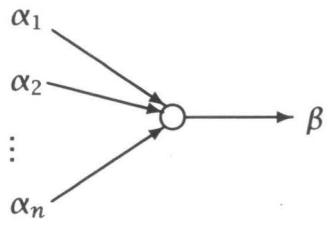
[1] Information about the Workflow Management Coalition can be found on `www.wfmc.org`.



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

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

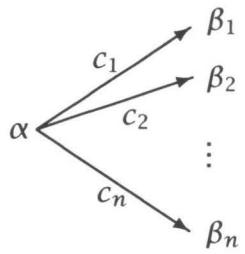
An AND-join specifies that a task may start only if a number of preceding tasks have all been completed. Here, β is started after $\alpha_1, \alpha_2, \ldots, \alpha_n$ have all been completed.



The condition for the start of task β is:

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

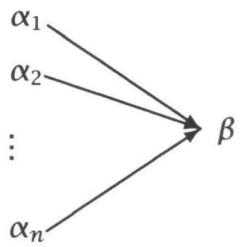
The previous two routing elements specified tasks that executed in parallel. We can also specify the selection of a subset of successor tasks, so that not all successor tasks need to be executed.



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

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

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

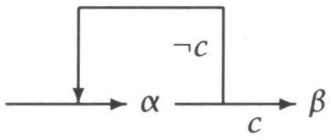


Hence, the set of predecessors of β needs not be a subset of the set of completed tasks:

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

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

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



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

To start α:

To start α:
$\alpha \in Completed(A) \wedge \neg c$
$\wedge$
$Inhib(\alpha) \cap Completed(A) = \emptyset$

On End(α)
if C
do Start(α)

To start β:
$\alpha \in Completed(A) \wedge c$
$\wedge$
$Inhib(\beta) \cap Completed(A) = \emptyset$

On End(α)
if $\neg C$
do Start(β)

□

8.3 Designing a workflow in DEGAS

In this section, we apply the design guidelines from Section 8.1 to the example of workflow management. The minimal capability and maximal encapsulation principle leads to a modularised approach to workflow management in active databases. Contrariwise, current approaches are all more or less global.

We show that the DEGAS approach to active database design leads to a clean database design for workflow management.

In the following discussion, we will first consider workflow *in abstracto*. Then, this discussion will be illustrated by a concrete example, which is an order processing flow. The billing task specified in the previous section is part of this order processing. This flow is depicted in Figure 8.3

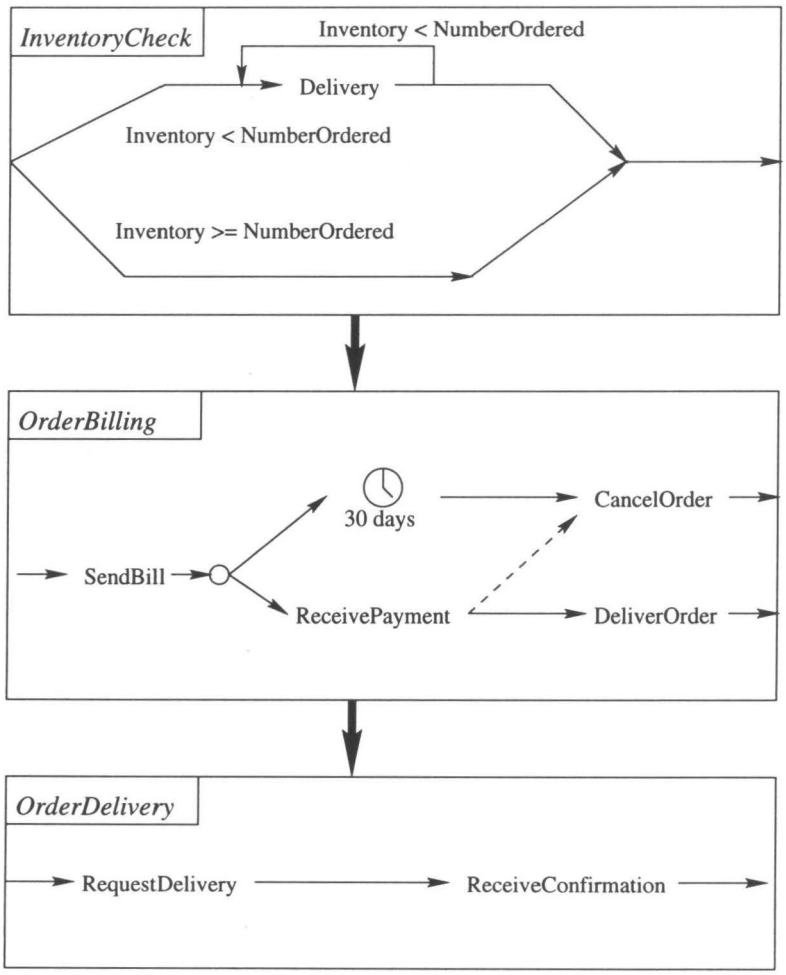


Figure 8.3: The workflow for order processing

8.3.1 Phase 1: Identifying the Objects

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

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

Object job
Attributes
item : string
number : integer
price : real
currentTask : number
...

The `agent` objects[2] implement the application functionality of the workflow. This means that they can represent anything from a piece of software processing the job, to an interface to a person. Thus, `agent` objects form the interface to the outside world. In our example, one of the agents is the Inventory Controller, which stores the number of items in stock and the reserved part of the stock.

Object InventoryControl
Attributes
Inventory : integer
Reserved : integer
...

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

The relations between these three classes of objects are mainly determined by their information exchange. In our example, we have three pairs of object classes, which means three potential binary relations. We briefly consider these three pairs. During the execution of a task of a job by an agent, the `agent`

[2]Not to be mistaken for any kind of *intelligent agents*, that will appear in Chapter 10.

object needs information from the job object. Hence, a job object has a relation with the agent object executing its current task. We call this relation the TaskExecution relation. The role of the agent is that of processor, while the job is processed. As a consequence of the minimality principle, explained in the previous section, this relation is only present while the agent executes the task. Once the task is completed, the relation is deleted again. In our example, the InventoryCheck relation is an example of a TaskExecution relation. Its only attributes are the partners in the relation, Job and Schema, that are specified in the **relation** clause. Other instances of TaskExecution in our example are the OrderBilling and OrderDelivery relations.

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

The remaining pair, schema and agent do not have a meaningful information exchange. The schema object contains information about the way jobs can be routed. This information is not necessary for an agent. Moreover, an agent can be used in multiple workflow schemas.

The result of this design phase in terms of generic workflow objects is depicted in Figure 8.4. Our concrete example is shown in Figure 8.5. Please note again, that the arrows do not imply any arity constraints on the relations. Instead, they point to the partner objects, on which the relation object depends for its existence. In DEGAS database design, the objects pointed to are specified earlier in the design iteration.

8.3.2 Phase 2: The actions in a workflow

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

Inherent actions of objects

Each agent has actions to start its task and to signal the completion of its task. Further actions of an agent are dependent on the kind of agent. For example, the InventoryControl agent has actions to reserve stock for an order and to put newly arrived stock in the inventory. The attributes increased in these methods are decreased, when the order is delivered in the OrderDelivery phase.

```
...
Methods
   reserve(number:integer) = {
```

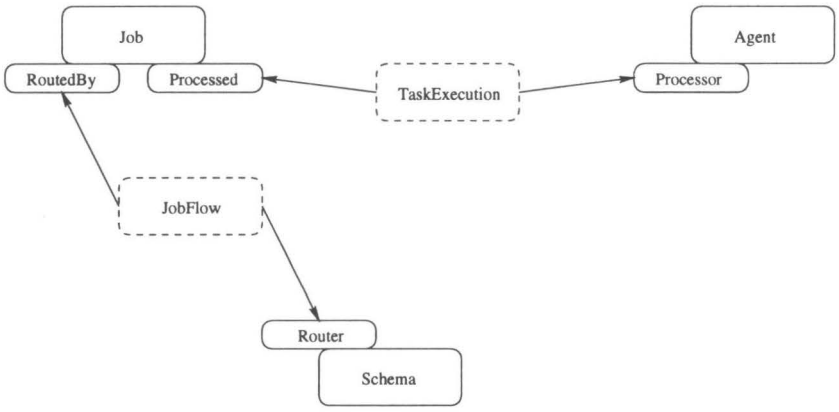


Figure 8.4: DEGAS object schema of a generic workflow

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

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

The services of a `schema` object are to provide information about the flow it defines. Its inherent action is the implementation of the *succ* function defined in Section 8.2. This means that it can answer the question, what comes after a specific task. The answer is provided by way of an addon, that implements the routing decision to be made after each task. Hence, the only information a job needs to provide to get an answer is the job it has just finished. This is a consequence of the minimality of information principle. For example, in the workflow shown in Figure 8.5, if a job has finished the `InventoryCheck`, it requests the next task from the `JobFlow` relation object. It forwards the request to the `Schema` object, that replies with the name of the addon implementing the `OrderBilling` phase

Transient actions of objects

Having defined the actions in the objects, we proceed to specify the actions in relations and their associated addons. With regard to a relation the most

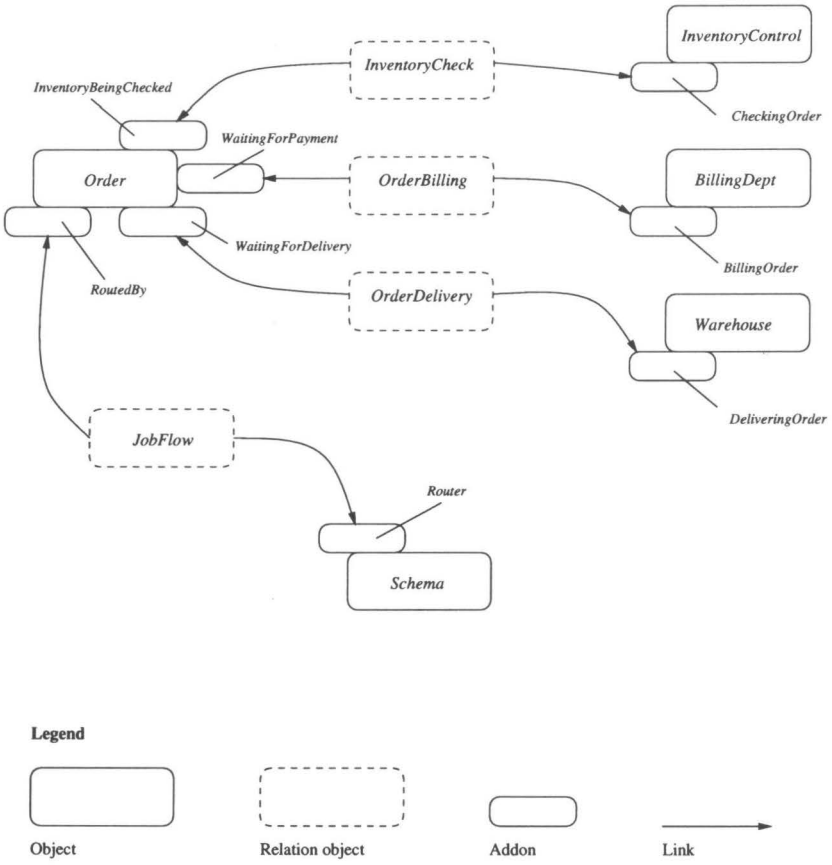


Figure 8.5: The object schema for an order processing workflow

important issue is, what a relation enables the partners to do. From this, we can derive the actions of the relation itself.

The `JobFlow` relation enables the `job` to follow a certain workflow. Through `JobFlow` it requests information on what task to execute next. The `RoutedBy` addon forwards a request for the next processing phase to the `JobFlow` relation object. The identity of the relation object is stored in the attribute `jobflow` of the `RoutedBy` addon.

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

The `JobFlow` relation object contains an action to inform the `job` partner of its next task. The answer is given by instructing the `job` object to extend itself with an addon that contains the actions of the next task. A further action is to replace the `Schema` object in the relation with a new object to facilitate schema evolution, which is discussed in Section 8.4.

```
Object JobFlow
...
Methods
   whatNext(task : string) = {
      succ = Schema.successor(task)
      Job.Extend(succ)
   }
   replaceSchema(newSchema:oid) = {
      Schema.remove(Router)
      Schema = newSchema
      Schema.Extend(Router)
   }
...
```

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

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

In our order processing example, the `InventoryCheck` relation is a specialisation of the `TaskExecution` relation. We first explain `TaskExecution` in general. A `TaskExecution` relation allows a `job` to be processed by an `agent`. The `job` is extended by a `Processed` addon. The `agent` is extended by a `Processor` addon. The existence of the relation means that the task will be executed, but it is not started immediately after the creation of the relation. Hence, the `Processed` addon contains the necessary actions to start the task. In addition, it contains actions that give the `agent` the information necessary to execute its task. The `Processor` addon contains the actions, that are specific to this task.

The `InventoryCheck` relation object implements the first processing phase which is done by the `InventoryControl` agent. The `InventoryCheck` relation object implements actions to start and to finish the processing phase.

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

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

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

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

```
Addon CheckingOrder
extends InventoryControl
...
Methods
```

```
    request(number : integer, dest : oid) = {
        noOfItems = number
    }
    getit(number : integer, dest : oid) = {
        reserve(number)
    }
...
```

8.3.3 Phase 3: Activation of action

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

Inherently, an object does not have relations with other objects. Hence, business rules are not inherently present. For our workflow application, there are no integrity constraints on the objects, `job`, `agent`, and `schema`. The inherent actions may be used, however, in the addons that extend objects.

The interaction scenario for the `JobFlow` relation is depicted in Figure 8.6. The `job` object, in our example the `order` object, requests the next task on completion of a task. The `whatNext` action of the `JobFlow` object gets this information by a call to the `successor` function of the `Schema` object. On receipt of the answer, the `JobFlow` object sends a message back to the `order` object. This is an `extend` action to add the addon implementing the next task, in this case the `OrderBilling` task.

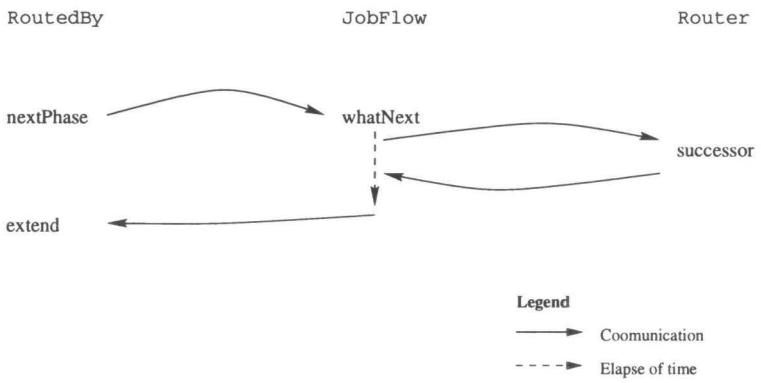


Figure 8.6: The interaction scenario for the `JobFlow` relation.

The interaction scenario for the InventoryCheck relation is shown in Figure 8.7. The goal of this task is to check, if the inventory suffices to deliver the order. The InventoryBeingChecked addon invokes the start action of the InventoryCheck object. At its turn, it sends a request message to the InventoryControl object. When the inventory check is successful, the getit action is executed by the InventoryControl object. This action invokes the Finish action of the InventoryCheck, which leads to execution of the End-InventoryCheck action of the order object.

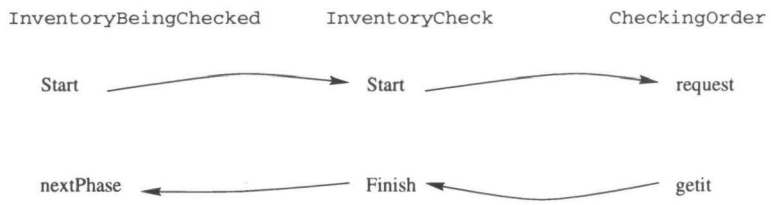


Figure 8.7: The interaction scenario for the InventoryCheck relation

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

This design phase leads to the activations of the different actions shown in Table 8.1.

8.3.4 Phase 4: Constraints on actions

The final phase of the design process is the specification of the order of the actions. This leads to a lifecycle for each object, that embodies its dynamic constraints. Please recall, that the lifecycle of an addon can include inherent actions of the object it extends.

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

```
...
Lifecycle
   Extend(RoutedBy);nextPhase*;Remove(RoutedBy)
...
```

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

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

Table 8.1: Activations of actions in the workflow example

```
...
Lifecycle
    Extend(InventoryBeingChecked);
        EndInventoryCheck;Remove(InventoryBeingChecked)
...
```

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

```
...
Lifecycle
    reserve*
    newstuff*
...
```

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

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

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

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

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

```
...
Lifecycle
    successor*
...
```

Complete Application This completes the design of our workflow example. We now give the complete DEGAS specifications of the objects discussed in this example.

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

Figure 8.8: DEGAS specification of the `order` object

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

Figure 8.9: DEGAS specification of the `InventoryBeingChecked` addon

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

Figure 8.10: DEGAS specification of the `InventoryCheck` relation object

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

Figure 8.11: DEGAS specification of the `InventoryControl` object

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

Figure 8.12: DEGAS specification of the `CheckingOrder` addon

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

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

Figure 8.13: DEGAS specification of the `RoutedBy` addon

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

Figure 8.14: DEGAS specification of the `Router` addon

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

Figure 8.15: DEGAS specification of the `JobFlow` relation object

8.4 Flexibility of the Workflow

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

8.4.1 Evolution of the Workflow Schema

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

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

An extensive discussion of schema evolution in workflow is found in [Casati *et al.*, 1996b]. The authors give a number of different policies to deal with activities in an evolving schema. The goal of these policies is to gracefully handle ongoing activities, that follow a schema being modified. In brief, the following policies are identified:

1. **Abort.** All activities following the old schema are aborted and restarted following the new schema.
2. **Flush.** Ongoing activities are completed according to the old schema, while new activities are started following the new schema.
3. **Progressive policies.** In these policies, ongoing activities are upgraded to a new schema without restarting.

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

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

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

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

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

All these approaches imply that a schema object, containing this transitional schema, is created. Since a job object does not contain any advance information about its routing, the change of schema can be enacted by a simple change of JobFlow relation. This is a distinct benefit of the DEGAS maximal encapsulation principle. Furthermore, the translation of workflow routing elements given in Section 8.2 makes the definition of a schema object a straightforward affair.

A final remark is, that the DEGAS approach also facilitates a truly ad-hoc way of dealing with schema evolution not mentioned by [Casati *et al.*, 1996b]. We can relate a job object to an interactive schema object, that prompts the workflow administrator for the next task on completion of each task in the activity. This might be useful for cases, where we only have a small number of job objects needing a transition schema.

8.4.2 Undoing tasks

In order to abort jobs or to apply progressive policies to jobs, we need the ability to roll back tasks. Aborting a task means that we have to reinstate the initial

state of the job object. Furthermore, some progressive policies may involve a partial roll-back to a previous state. Here, we look at the problem of rolling back tasks in the workflow design discussed in this chapter.

As we explained in Section 5.4, previous states of a DEGAS object are stored as part of its history, together with the actions that brought the object into that state. Hence, all information to bring back the job object to its original state is found in the object itself. Rolling back the actions with regard to the tasks that were executed, is basically a question of removing these actions from the history of the object. Since the current state is the latest state in the history, the current state of the object will then be automatically set to the state before execution of the rolled-back action.

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

Compensating tasks are analogous to the concept of compensating transactions used for sagas [Garcia-Molina and Salem, 1987]. A saga consists of a sequence of sub-transactions $T_1, \ldots, T_n$. It is either completely executed, or completely undone. Suppose that for each sub-transaction T_i we have a compensating transaction C_i. Then, we execute either $T_1; T_2; \ldots; T_n$ or $T_1; \ldots; T_j; C_j; \ldots; C_1$. Instead, we define a partial order on tasks, that may be extended. Thus, we do not need to roll back a transaction completely to its start, but we can roll back only part of its actions. Hence, we relax the requirement on a saga. Suppose we have a partially ordered set of tasks T, where each $\tau_i \in T$ has an associated compensating action ρ_i. In addition, we have a function $Compensate(\tau)$, that yields the compensating task of task τ. Please note that a task undoes its compensating task, so the compensation of a task itself is its compensating task and $Compensate(Compensate(\tau)) = \tau$. Then, we have the following requirements on two subsequent tasks:

$$
\begin{aligned}
\tau_i;\tau_j \;&:\; \tau_i < \tau_j \wedge \nexists \psi \in T : \tau_i < \psi < \tau_j \\
\tau_i;\rho_j \;&:\; \rho_j = Compensate(\tau_i) \\
\rho_i;\rho_j \;&:\; Compensate(\rho_i) > Compensate(\rho_j) \\
&\quad \wedge \nexists \psi \in T : Compensate(\rho_i) > \psi > Compensate(\rho_j) \\
\rho_i;\tau_j \;&:\; \exists \psi \in T : \psi < Compensate(\rho_i) \wedge \psi < \tau_j \\
&\quad \wedge \nexists \phi \in T : \phi > \psi \wedge \phi < Compensate(\rho_i) \wedge \phi < \tau_j
\end{aligned}
$$

8.4.3 Changing Task Execution

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

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

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

8.5 Conclusion

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

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

Chapter 9

(Un)decidability Results for DEGAS Objects

The first phase of information systems design is the specification of a model, which was discussed in the previous chapter. After this, the design must be verified against the user and system requirements. The verification of user requirements has given rise to the research area of requirements engineering [Wieringa, 1996, IEEE, 1994]. However, we do not discuss it here. Instead, we focus on the verification of desirable system properties. In particular, we examine system properties specific to active databases.

The additional functionality of an active database leads to new aspects in the behaviour of such a system. These are caused by the interactions within a set of rules. For example, members of the set may mutually activate or deactivate each other. Because the autonomous nature of DEGAS objects, their behaviour is controlled through their definition only. A *design theory* defines properties on sets of rules and, if possible, provides algorithms to detect such properties.

Properties of rule sets studied in the literature are *termination* and *confluence*. A set of rules terminates if, starting with any initial database state, the selectors of all the rules become false in a finite number of steps. That is, the database converges to a final state. A terminating set of rules is called confluent, if the final state is determined completely by the initial state and the rule set.

As was discussed in Section 6.2.4, the rule semantics of DEGAS are instance-oriented. In order to generalise the results of this chapter, we also look into rules executing under set-oriented semantics. Recall from Section 3.1, that under instance-oriented semantics a rule executes, non-deterministically, on *one* object that satisfies this condition. Under set-oriented semantics, a rule executes simultaneously on *all* objects that satisfy its selector. In this chapter, we shall abbreviate the terms to set semantics and instance semantics.

The two types of semantics suggest a third property, which we call *indifference*, that relates the two semantics. A rule set that is confluent under both semantics is indifferent, if the unique state under the two semantics is identical. This property is of interest for comparing the execution of a rule set in DEGAS with its execution in another system.

The development of a design theory for rules has been advanced by targeting on either *sufficient conditions* or on *decidability*. Examples of the former approach are [Simon and de Maindreville, 1988, Aiken *et al.*, 1992]. In the context of the RDL rule system, [Simon and de Maindreville, 1988] formulates a condition under which set and instance based rule execution coincide. [Widom and Finkelstein, 1990] and [Widom *et al.*, 1991] defined a production rule language for the Starburst database system. In [Aiken *et al.*, 1992] sufficient conditions for both termination and confluence of these production rules are formulated. An important foundation of our results is the work on decidability reported in [Voort, 1994]. A property is called decidable, if there exists an algorithm that, given a set of rules as input, decides in finite time, whether this set satisfies the property or not. Examples of this approach are [Abiteboul and Simon, 1991], [Voort, 1994] and this chapter.

Due to the complexity of the subject, we use a restricted DEGAS model in this chapter, named DEGAS⁻. The main restriction is the omission of events. Furthermore, the action of a rule is restricted to the modification of an object's attributes. In this chapter, we show that termination and confluence are already undecidable for very limited rule models. These properties are decidable for an even more restricted DEGAS^{-2} model, that contains rules with local conditions and constant assignment only. The addition of path expressions to conditions in DEGAS⁻ makes it possible to emulate a Turing Machine. Hence, termination and confluence are undecidable in DEGAS⁻.

9.1 The DEGAS⁻ Model

In this section, we define the DEGAS⁻ model, that is a restricted DEGAS model. The restrictions are the following:

1. Event specifications are omitted from rules.
2. Lifecycles are omitted.
3. Relation objects and addons are not considered.
4. Methods can only make assignments.
5. The only types for attributes are integer and object id.

In order to compare set and instance semantics, the semantics of the DEGAS⁻ model are defined different from the semantics of DEGAS in Section 5

9.1.1 Syntax

For clarity, we give the full syntax definition of the DEGAS⁻ model in Figure 9.1 and Figure 9.2. It is a subset of the DEGAS syntax as given in Section 4.3. Uniqueness constraints and referential constraints apply, like in the full DEGAS model.

⟨Class⟩	→	`Object` ⟨ClassName⟩	(9.1)
		⟨AttributeSection⟩	
		⟨MethodSection⟩	
		⟨RuleSection⟩	
		`EndClass`	
⟨AttributeSection⟩	→	`Attributes`	(9.2)
		⟨AttributeList⟩	
⟨AttributeList⟩	→	⟨AttributeDecl⟩	(9.3)
		| ⟨AttributeDecl⟩ , ⟨AttributeList⟩	
⟨AttributeDecl⟩	→	⟨AttributeName⟩ : ⟨Type⟩	(9.4)
⟨Type⟩	→	`Integer`	(9.5)
⟨Type⟩	→	⟨ClassName⟩	(9.6)
⟨MethodSection⟩	→	`Methods`	(9.7)
		⟨MethodList⟩	
⟨MethodList⟩	→	⟨MethodDecl⟩	(9.8)
		| ⟨MethodDecl⟩ , ⟨MethodList⟩	
⟨MethodDecl⟩	→	⟨MethodName⟩ (⟨ParameterList⟩)= {	(9.9)
		⟨StatementList⟩	
		}	
⟨StatementList⟩	→	⟨Statement⟩	(9.10)
		| ⟨Statement⟩ ; ⟨StatementList⟩	
⟨Statement⟩	→	⟨AttributeName⟩ := ⟨BasicExpression⟩	(9.11)
⟨BasicExpression⟩	→	0 | 1 | ...	(9.12)
		| ⟨AttributeName⟩	
⟨ActParamList⟩	→	⟨ActParam⟩	(9.13)
		| ⟨ActParam⟩ , ⟨ActParamList⟩	
⟨ActParam⟩	→	⟨ParameterId⟩ = ⟨Expression⟩	(9.14)

Figure 9.1: The DEGAS⁻ syntax.

An example DEGAS⁻ object is given below. It has a method `multiply_no` that does what its name suggest. Furthermore, it contains a rule, that selects cells

⟨RuleSection⟩	→	`Rules` ⟨RuleList⟩	(9.15)
⟨RuleList⟩	→	⟨Rule⟩ | ⟨Rule⟩ , ⟨RuleList⟩	(9.16)
⟨Rule⟩	→	`if` ⟨SelectionCondition⟩ `do` ⟨Action⟩	(9.17)
⟨Action⟩	→	⟨MethodCall⟩	(9.18)
⟨SelectionCondition⟩	→	⟨SelectionCondition⟩ ∧ ⟨SelectionCondition⟩	(9.19)
⟨SelectionCondition⟩	→	⟨SelectionCondition⟩ ∨ ⟨SelectionCondition⟩	(9.20)
⟨SelectionCondition⟩	→	⟨SelectExpr⟩ = ⟨SelectExpr⟩	(9.21)
⟨SelectExpr⟩	→	⟨AttributeID⟩	(9.22)
⟨SelectExpr⟩	→	0 | 1 | ...	(9.23)
⟨SelectExpr⟩	→	⟨SelectExpr⟩ . ⟨SelectExpr⟩	(9.24)

Figure 9.2: The DEGAS⁻ syntax continued.

with ten as a value of no and multiplies it by two.

```
Object cell
Attributes
   no: Integer
   neighbour : cell
Methods
   multiply_no(factor:Integer) = {
      no:=no*factor
   }
Rules
   If no=10
   do multiply_by(2)
EndObject
```

9.1.2 Semantics

The semantics of the DEGAS⁻ model are defined differently than the semantics of the full DEGAS model in order to enable a comparison between instance semantics and set semantics for rule execution. This allows us to generalise the results of this chapter to active databases in general, in addition to DEGAS. Since the semantics of DEGAS are object-centered, both instance semantics and set semantics make little sense.

The DEGAS⁻ semantics assigns objects and values to the database as a whole.

An instance of a database is determined by an extension and an interpretation. The extension assigns objects to each class in the database. The interpretation assigns values to the attributes.

The extension of a database assigns a set of objects to each class. An object is identified by a unique object identifier. Therefore, we assume the existence of a set of object identifiers Oid. The extension function then assigns a subset of Oid to each class, such that all object identifiers are assigned to one class only.

Definition 60 *Let H be a* DEGAS⁻ *schema, i.e., a set of* DEGAS⁻ *object definitions. An extension $Ext : H \longrightarrow \mathcal{P}Oid$ assigns to each class of H a set of objects such that if $C_1, C_2 \in H$ and $C_1 \neq C_2$, then $Ext(C_1) \cap Ext(C_2) = \emptyset$.*

The type Integer has the obvious extension:

Definition 61 *The extension of the type Integer is the set of natural numbers $\mathbb{N}$.*

The extension of the database gives us the sets of objects in each class. The contents of the objects are defined by the interpretation, that assigns values to all the attributes in the database. Please note, that a DEGAS⁻ interpretation assigns values to a complete database at once contrary to the object-centered approach of a DEGAS interpretation. The DEGAS⁻ interpretation gives a table for every attribute in the database schema. This table has two columns, the first containing object identifiers, and the second the value of the attribute in the object represented by this identifier. An example is the following interpretation for the attribute `no` of the class `cell` defined above.

Oid	Value
345	3
874	25
902	16

In this example, we have three objects. The value of the attribute `no` for object 874 is 25. The interpretation of an attribute is a function from the extension of the class it belongs to, to the extension of the type of the attribute. The extension of the type itself is either a class extension, or the extension of the type Integer.

Definition 62 *Let Ext be an extension of a database schema H. An interpretation I for Ext and H assigns to each type declaration $a : \tau$ a function $I(a : \tau) : Ext(C) \longrightarrow Ext(\tau)$.*

A database for a schema is a pair of an extension and an interpretation.

Definition 63 *A database for a schema H is a pair (Ext, I) where Ext is an extension for H and I an interpretation for Ext and H. The universe of all databases is denoted by DB_H. Individual database states are denoted by db, db_1, db_2, ...*

The semantics of method execution in DEGAS⁻ relates two interpretations to each other. To be precise, we describe the interpretation of the database after the method execution in terms of the interpretation before the method execution. This is captured by the notion of a variant interpretation. A variant of an interpretation I is denoted by $I\{v/(a : \tau)(o)\}$. The variant is the same as I, except when $I(a : \tau)$ is applied to the object o, where it yields v. A property of variants is the independence of variants on different objects, which we state without proof.

Proposition 2 *Given an extension Ext and an interpretation I for a DEGAS⁻ schema. Then:*

$$\begin{aligned}
&\forall o_1, o_2 \in Ext : \\
&o_1 \neq o_2 \\
&\Rightarrow \\
&I\{v_1/(a:\tau)(o_1)\}\{v_2/(a:\tau)(o_2)\} \\
&\quad = I\{v_2/(a:\tau)(o_2)\}\{v_1/(a:\tau)(o_1)\}
\end{aligned}$$

The function M that defines the semantics of method execution gives us a new interpretation of the database, that is variant on the attributes modified. It is defined as follows:

Definition 64 *Given a method of class $C \in H$:*

$$\begin{aligned}
&m(l_1 : \tau_1, \ldots, l_n : \tau_n) = \{ \\
&\quad a_i := l_i \\
&\quad succ(b_j) \\
&\}
\end{aligned}$$

for $i = 1 \ldots n, j = 1 \ldots m$, where:

$$\begin{aligned}
&\forall i \in \{1, \ldots, n\} : \text{“}a_i : \tau_i\text{”} \in Attr(C) \\
&\forall j \in \{1, \ldots, m\} : \text{“}b_j : \sigma_j\text{”} \in Attr(C) \\
&\{\text{“}a_1 : \tau_1\text{”}, \ldots, \text{“}a_n : \tau_n\text{”}\} \cap \{\text{“}b_1 : \sigma_1\text{”}, \ldots, \text{“}b_m : \sigma_m\text{”}\} = \emptyset
\end{aligned}$$

If o is an object in $Ext(C)$ and $m(l_1 = v_1, \ldots, l_n = v_n)$ a correct method call, then the function M is defined for the execution of m by o in the database (Ext, I) as:

$$\begin{aligned}
&M(m(l_1 = v_1, \ldots, l_n = v_n)(o)(Ext, I)) = \\
&\quad (Ext, I\{v_1/(a_1 : \tau_1)(o)\}, \ldots, \{v_n/(a_n : \tau_n)(o)\}, \\
&\qquad \{(I(b_1 : \sigma_1) + 1)/(b_1 : \sigma_1)(o)\}, \\
&\qquad \ldots, \\
&\qquad \{(I(b_m : \sigma_m) + 1)/(b_m : \sigma_m)(o)\})
\end{aligned}$$

9.1.3 The DEGAS^{-2} Model

In our discussion of decidability results, we first consider a restriction of DEGAS⁻, which is named DEGAS^{-2}. We restrict the selection condition of a query to local attributes only, which means that production 9.24 is omitted from the syntax definition. An example of a class definition DEGAS^{-2} is:

```
Object cell
Attributes
   no: Integer
   neighbour : cell
Methods
   new_value(number:Integer) = {
      no:=number
   }
Rules
   If no=10
   do new_value(5)
EndObject
```

9.1.4 Rule Semantics

Rules can be applied to a database in two different ways [Simon and de Maindreville, 1988]. This results in two different semantics for rule execution. With set semantics the action is executed on all objects satisfying the rule condition simultaneously. Instance semantics means that the action is executed one at the time on these objects. First, we define the set of object satisfying a rule's selection condition. In the following definition, the result of rule R applied to a database db under set semantics is denoted by $M(s)(R, db)$. Under instance semantics, this is denoted by $M(i)(R, db)$. The execution of a rule, like a method execution, changes the interpretation of the database. Therefore, we can define the resulting interpretation in terms of the variants induced by the rule.

Definition 65 *Let $R = (Q, M(l_1 = b_1, \ldots, l_n = b_n))$ be a trigger in a schema H with M defined as $M(l_1 : \tau_1, \ldots, l_n : \tau_n) = \{a_1 := l_1; \ldots; a_n := l_n\}$. Furthermore, we have:*

$db = (Ext, I)$	*a database for H.*
$Select((Ext, I), R)$	*the set of objects in db satisfying the condition of rule R.*
$Pick : \mathcal{P}Oid \longrightarrow Oid$	*a function that arbitrarily selects an object from a set of objects.*
$Var(o_i)$	*the variant of o_i induced by trigger R.*

The execution of R under set semantics is defined by:

$$M(s)(R, (Ext, I)) = \begin{cases} (Ext, I\{Var(o_1) \ldots Var(o_m)\} \\ \quad \text{if } Select((Ext, I), R) = \{o_1, \ldots, o_m\} \\ (Ext, I) \\ \quad \text{if } Select((Ext, I), R) = \emptyset \end{cases}$$

The execution of R under instance semantics is defined in two phases in order to

separate out the random selection of an object to execute on.

$$M(i)(R,(Ext,I)) = \begin{cases} M_i' \circ Pick(Select((Ext,I),R)) \\ \quad if\, Select((Ext,I),R) \neq \emptyset \\ \quad where\, M_i'((R,o),(Ext,I)) = (Ext, I\{Var(o)\}) \\ M(i)(R,(Ext,I)) = (Ext,I) \\ \quad if\, Select((Ext,I),R) = \emptyset \end{cases}$$

The above definition defines the result of a single rule application. If a set of rules is present on the database, we have an execution cycle. This cycle executes as long as there are rules applicable to the database. During the cycle one of the set of applicable rules is randomly chosen for execution and executed. After that the next iteration starts. Please note, that the DEGAS⁻ execution cycle is defined for the complete database, contrary to the execution cycle of a single DEGAS object.

Definition 66 *Let $\mathcal{R}$ be a set of rules of the form $R = (Q_R, M_R)$ with an initial database db. Then the behaviour of $\mathcal{R}$ under semantics sem is defined by:*

```
Execute(R,db,sem) {
  While ∃R ∈ R : Ext_db(Q_R) ≠ ∅ do
    T := choose({R|R ∈ R ∧ Ext_db(Q_R) ≠ ∅)
    db := M(sem)(R,db)
  od
  return db
}
```

The process `Execute(`$\mathcal{R}$`,db,sem)` induces a set of execution sequences. An execution sequence gives a trace of rule execution. In the case of set semantics, it is a sequence of rules. In the case of instance semantics, we also include the information, which object the rule was executed on. A sequence is defined as a function from the set of natural numbers to the set of rules. The function assigns a rule to each position in the sequence. If the function is total, i.e., it assigns a rule to each position, then it is a sequence.

Definition 67 *$\mathcal{R}$ is a set of rules for a schema H. If $Sq : \mathcal{N}^+ \rightarrow \mathcal{R}$ is a partial function whose support is a contiguous set starting at 1, then $Sq \in Seq(s)$, $length(Sq) = |Support(Sq)|$. Sq is written as a list $[Sq(1);\ldots;Sq(n)]$.*

The ith element of a sequence Sq, denoted by Sq_i, is a rule R_i

Under instance semantics, a sequence also states to which object a rule is applied at each place in the sequence.

Definition 68 *$\mathcal{R}$ is a set of triggers for a schema H. If $Sq : \mathcal{N}^+ \rightarrow \mathcal{R} \times Oid$ is a partial function whose support is a contiguous set starting at 1, then $Sq \in Seq(i)$, $length(Sq) = |Support(Sq)|$. Sq is written as a list $[Sq(1);\ldots;Sq(n)]$.*

Here, Sq_i is a pair of a rule and an object (R_i, o_i).

The execution of a rule sequence is defined inductively in the following way:

Definition 69 *$\mathcal{R}$ is a set of triggers for a schema H. The execution of a sequence $Sq \in Seq(s)$ on a database db, where $Seq(s)$ is the set of sequences over $\mathcal{R}$ under set semantics, is defined as:*

1. *if* $Sq \equiv [Sq(1);\ldots;Sq(n)]$
   then $M(s)(Sq, db) = M(s)([Sq(2);\ldots;Sq(n)], M(s)(Sq(1), db))$

2. *if* $Sq \equiv [Sq(1);Sq(2);\ldots]$
   then $M(s)(Sq, db) = M(s)([Sq(2);\ldots], M(s)(Sq(1), db))$

The execution of a sequence $Sq \in Seq(i)$ on a database db, where $Seq(i)$ is the set of sequences over $\mathcal{R}$ under instance semantics, is defined as:

1. *if* $Sq \equiv [Sq(1);\ldots;Sq(n)]$
   then $M(i)(Sq, db) = M(i)([Sq(2);\ldots;Sq(n)], M'(i)(Sq(1), db))$

2. *if* $Sq \equiv [Sq(1);Sq(2);\ldots]$
   then $M(i)(Sq, db) = M(i)([Sq(2);\ldots], M(i)'(Sq(1), db))$

A valid execution sequence must satisfy a number of requirements. A sequence $[Sq(1);\ldots;Sq(n)]$ is an execution sequence, if each rule $Sq(i+1)$ is activated after the execution of $[Sq(1);\ldots;Sq(i)]$. If the sequence is finite, no rules are activated after the last rule of a sequence.

Definition 70 *Let $\mathcal{R}$ be a set of rules for a schema H, let db be a database, and let $Seq(s)$ and $Seq(i)$ be the sets of sequences over $\mathcal{R}$. The set $Seq(s)(\mathcal{R}, db)$ of execution sequences over $\mathcal{R}$ is db under set semantics is defined as follows:*

if $Sq \in Seq(s)$
$\wedge$
$Select(db, Sq(1)) \neq \emptyset$
$\wedge$
$\forall i \in \{2,\ldots,n\}:$
$\quad Select(M(s)([Sq(1);\ldots;Sq(i-1)]), Sq(i)) \neq \emptyset$
$\wedge$
$\forall R \in \mathcal{R}:$
$\quad Select(M(s)(Sq, db), R) = \emptyset$
then $Sq \in Seq(s)(\mathcal{R}, db)$

The set of execution sequences under instance semantics $Seq(i)(\mathcal{R}, db)$ is de-

fined analogously as follows:

$$
\begin{array}{ll}
\textit{if} & Sq \in Seq(i) \\
& \wedge \\
& o_i \in Select(db, Sq(1).R) \\
& \wedge \\
& \forall i \in \{2, \ldots, n\} : \\
& \quad o_i \in Select(M(i)([Sq(1); \ldots; Sq(i-1)]), Sq(i).R) \\
& \wedge \\
& \forall R \in \mathcal{R} : \\
& \quad Select(M(i)(Sq, db), R) = \emptyset \\
\textit{then} & Sq \in Seq(i)(\mathcal{R}, db)
\end{array}
$$

9.2 Predicates

In the previous section, we defined the DEGAS$^-$ model's syntax and semantics. In this section, we address the predicates on rule sets that we examine in this chapter.

9.2.1 Termination

Termination means that all executions of a rule set terminate on all possible database states. Hence, all execution sequences on all databases must be finite.

Definition 71 (Termination) *Let $\mathcal{R}$ be a set of rules for a schema H. Let sem denote either set or instance semantics.*

$$
\begin{array}{l}
Terminate(\mathcal{R}, sem) \stackrel{def}{=} \\
\quad \forall db \in DB_H, \forall Sq \in Seq(sem)(\mathcal{R}, db), \exists n \in \mathcal{N}^+ : \\
\quad\quad length(Sq) = n
\end{array}
$$

9.2.2 Confluence

Confluence means that all possible executions of a rule set yield the same final database state. Because a non-terminating execution does not yield a final database state, a preliminary requirement for confluence is, that the rule set is terminating. In terms of execution sequences, confluence means that the resulting database state is invariant under the choice of an execution sequence.

Definition 72 (Confluence) *Let $\mathcal{R}$ be a set of triggers for a schema H. Let sem denote either set or instance semantics.*

$$
\begin{array}{l}
Confluent(\mathcal{R}, sem) \stackrel{def}{=} \\
\quad \forall db \in DB_H, \forall Sq \in Seq(sem)(\mathcal{R}, db) : \\
\quad\quad M(sem)(Sq_1, db) = M(sem)(Sq_2, db) \\
\quad \wedge \\
\quad Terminate(\mathcal{R}, sem)
\end{array}
$$

9.2.3 Termination in n steps

In most rule models, termination is an undecidable property. Therefore, we are interested in a stronger predicate than termination, viz., termination in a certain number of steps. This predicate is stronger than termination, because a rule set not terminating in n steps might terminate in $n + 33$ steps.

To define termination in n steps, we need to define a step. Under set semantics, we simply take one execution of a rule as one step. We cannot do this under instance semantics, since a rule application only executes on one object at the time. Because we do not wish to make our choice of n dependent of the size of the database, we count the number of rule applications to one object. Thus, under set semantics n denotes the maximum number of times a rule may execute. Under instance semantics, n denotes the maximum number of times a trigger may execute on one object.

Definition 73 *If $\mathcal{R}$ is a set of rules for a schema H and n a natural number, then*

$$
\begin{aligned}
&Terminate(n, \mathcal{R}, s) \stackrel{def}{=} \\
&\quad \forall db \in DB_H, \forall Sq \in Seq(s)(\mathcal{R}, db), \forall R \in \mathcal{R}: \\
&\qquad |\{i | Sq(i) = R\}| \leq n \\
&Terminate(n, \mathcal{R}, i) \stackrel{def}{=} \\
&\quad \forall db \in DB_H, \forall o \in Ext_{db}, \forall Sq \in Seq(i)(\mathcal{R}, db), \forall R \in \mathcal{R}: \\
&\qquad |\{i | Sq(i) = (R, o)\}| \leq n
\end{aligned}
$$

The execution sequences are finite, because all rule sets and all databases are finite. Thus, rule sets that terminate in n steps terminate.

Proposition 3 *If $\mathcal{R}$ is a set of rules for a schema H and sem denotes either set or instance semantics, then*

$$Terminate(n, \mathcal{R}, sem) \Longrightarrow Terminate(\mathcal{R}, sem)$$

Proof Obvious. □

9.2.4 Independence

Like we defined termination in n steps as a stronger alternative for termination, we can define a stronger alternative for confluence. This stronger predicate is independence. Instead of looking at a complete rule set, we examine a pair of rules at the time. A pair is said to be independent, if the two rules commute. This means that the result of their execution is the same for both possible orders of execution. A set of rules is independent, if all pairs in the set commute.

Definition 74 *If $\mathcal{R}$ is a set of triggers for a schema H, then:*

$$
\begin{aligned}
&Independent(\mathcal{R}, set) \stackrel{def}{=} \\
&\quad \forall R_i, R_j \in \mathcal{R}, \forall db \in DB_H : \\
&\qquad M(s)(R_i, M(s)(R_j, db)) \\
&\qquad = \\
&\qquad M(s)(R_j, M(s)(R_i, db))
\end{aligned}
$$

$$
\begin{aligned}
&Independent(\mathcal{R}, instance) \stackrel{def}{=} \\
&\quad \forall R_i, R_j \in \mathcal{R}, \forall db \in DB_H, \forall o_k, o_l \in Ext_{db} : \\
&\qquad M(i)(R_i(o_k), M(i)(R_j(o_l), db)) \\
&\qquad = \\
&\qquad M(i)(R_j(o_l), M(i)(R_i(o_k), db))
\end{aligned}
$$

A useful property of an independent rule set is, that execution sequences can be rearranged. This comes in handy in a number of proofs. An example is the proof that independence implies confluence for terminating rule sets [Aiken *et al.*, 1992].

Proposition 4 *If $\mathcal{R}$ is a set of rules and sem denotes either instance or set semantics, then:*

$$
\begin{aligned}
&Terminate(\mathcal{R}, sem) \wedge Independent(\mathcal{R}, sem) \\
&\Longrightarrow \\
&Confluent(\mathcal{R}, sem)
\end{aligned}
$$

Proof We have to prove that:

$$
\begin{aligned}
&\forall db \in DB_H, \forall Sq_1, Sq_2 \in Seq(sem)(\mathcal{R}, db) : \\
&\quad M(sem)(Sq_1, db) = M(sem)(Sq_2, db)
\end{aligned}
$$

Since both Sq_1 and Sq_2 are execution sequences, no rule is activated after their execution. Thus, for all database states

$$M(sem)(Sq_1; Sq_2, db) = M(sem)(Sq_1, db)$$

and likewise

$$M(sem)(Sq_2; Sq_1, db) = M(sem)(Sq_2, db)$$

With all rules pairwise independent, we can rearrange $Sq_2; Sq_1$ into $Sq_1; Sq_2$. Thus, we have

$$M(sem)(Sq_2; Sq_1, db) = M(sem)(Sq_1; Sq_2, db)$$

and

$$M(sem)(Sq_1, db) = M(sem)(Sq_2, db)$$

□

9.2.5 Decidability

The last definition needed in this chapter is that of decidability. For this, we use the standard notion of the existence of a decision procedure (see, e.g., [Lewis and Papadimitriou, 1981]).

Definition 75 *A predicate is decidable, iff there exists an algorithm that on all possible input:*

1. *terminates*
2. *on termination gives the correct answer with regard to the truth of the predicate relative to the input.*

9.3 Decidability Results for DEGAS^{-2}

In this section, we discuss decidability of rule predicates in the DEGAS^{-2} model. Recall, that this model only allows local conditions and replacement of attributes by constants. As can be expected in such a simple model, both termination and confluence are decidable properties of rule sets in DEGAS^{-2}.

We first look at termination and confluence of a single rule. Since the effect of a rule's action is idempotent, these are decidable predicates.

Theorem 6 *In the* DEGAS^{-2} *model, given a singleton rule set* $\mathcal{R} = \{R\}$, $R = (Q_R, M_R)$ *and semantics* sem. *In this case, we have the decidable predicates* $Terminate(\mathcal{R}, sem)$ *and* $Confluent(\mathcal{R}, sem)$.

Proof To prove the decidability of termination and confluence, we show that we can construct a finite database state to represent all possible database states, as was first explained in [Voort, 1994]. This database state is called the *typical database state*. The typical database state is constructed relative to a rule definition. From the attributes of the class schema, the constants in the selection conditions and the methods, we construct a finite set of partition conditions. Every object in all of the possible database states satisfies one of these conditions. In addition, method execution gives a uniform transition between these conditions. Based on this knowledge, we construct a graph that enables us to decide termination and confluence.

We start by defining a set EC of elementary conditions on the attributes. Let A be the set of all attributes in the class on which R is defined. C_R is the set of all constants appearing in Q_R and M_R. The set of elementary conditions EC with regard to R is defined by the following grammar:

$$\langle \text{Econd} \rangle \longrightarrow \langle \text{Attr} \rangle = \langle \text{BasicExpr} \rangle$$
$$\langle \text{BasicExpr} \rangle \longrightarrow \langle \text{Attr} \rangle \mid \langle \text{Const} \rangle$$

where the non-terminal ⟨Attr⟩ yields all elements of A and ⟨Const⟩ yields all elements of C_R.

Obviously EC does not take the types of attributes and constants into account. Therefore, we restrict EC to the set of well-typed elementary conditions WEC as follows:

$$\begin{aligned}&\forall x:\tau, y:\tau:\\ &\quad x = y \in EC\\ &\quad \Rightarrow\\ &\quad x = y \in WEC\end{aligned}$$

The elementary conditions only express equalities of one or two attributes at a time. To be able to express arbitrary conditions on an object, we obtain all composite conditions in the set $Cond$.

$$\begin{aligned}&\text{If } c \in WEC \text{ then } c \in Cond\\ &\text{If } c_1, c_2 \in Cond \text{ then } c_1 \wedge c_2 \in Cond\end{aligned}$$

This definition yields a set that also contains inconsistent conditions. We are, however, able to decide what conditions are consistent. This is stated by the following claim, that is proven below.

Claim 6.1 *Determining the consistency of a condition $\phi \in Cond$ is decidable. If ϕ is consistent, we can construct a database state that contains an object satisfying ϕ.*

Knowing that the consistency of a condition is decidable, we can restrict our conditions to the set $CCond$ of consistent conditions.

$$CCond = \{c \in Cond | c \text{ is consistent}\}$$

Multiple conditions are satisfied by an object, because most conditions do not take all attributes of an object into account. To characterise an object, we want those conditions that specify equalities of all attributes. To that end, we define a partial order on $CCond$:

$$\begin{aligned}&\forall \phi, \psi \in CCond, \phi = \phi_1 \wedge \ldots \wedge \phi_k, \psi = \psi_1 \wedge \ldots \wedge \psi_l:\\ &\quad \forall i \in \{1 \ldots k\} \exists j \in \{1 \ldots l\} : \phi_i \Rightarrow \psi_j\\ &\quad \Longrightarrow\\ &\quad \psi > \phi\end{aligned}$$

It is easy to see that this defines a partial order. In this order, the maxima are those conditions that incorporate all attribute-attribute and attribute-constant relations. The existence of these maxima follows from the finite size of the attribute set, implying the finite size of $CCond$. Therefore, we use these conditions as partitioning conditions.

$$PCond = \{c \in CCond | c \text{ is maximal}\}$$

The partition conditions characterise all possible objects in all possible databases relative to this rule. Uniform transitions exist between partition conditions to represent the effect of method execution. These two properties are expressed in the following claim:

Claim 6.2 *Partition conditions satisfy the following properties:*

1. $\forall db \in DB, \forall o \in db :$
   $\exists! pc \in PCond : pc(o, db)$

2. $\forall db_1, db_2 \in DB, \forall o_1 \in db_1, \forall o_2 \in db_2 :$
   $pc(o_1, db_1) \wedge pc(o_2, db_2)$
   $\Longrightarrow$
   $\exists pc' \in PCond :$
   $pc'(o_1, Execute(R, db_1, set))$
   $\wedge$
   $pc'(o_2, Execute(R, db_2, set))$

   under set semantics and

   $\forall db_1, db_2 \in DB, \forall o_1 \in db_1, \forall o_2 \in db_2 :$
   $pc(o_1, db_1) \wedge pc(o_2, db_2)$
   $\Longrightarrow$
   $\exists pc' \in PCond :$
   $pc'(o_1, Execute(R(o_1), db_1, instance))$
   $\wedge$
   $pc'(o_2, Execute(R(o_2), db_2, instance))$

   under instance semantics.

where $pc(o, db)$ denotes that pc is a partitioning condition characterising o in db.

Claim 6.1 said that for each consistent condition we can construct a database state satisfying that condition. Because we may assume without loss of generality that object identifiers are unique over all db_ϕ, we can construct a database state $db = \bigcup_{\phi \in PCond} db_\phi$, that is typical, i.e., $\forall c \in PCond, \exists o \in db : c(o, db)$.

We have shown that partition conditions represent all possible object states. Further, we have shown that the effect of method application is uniformly represented by a transition from one partition condition to another. We now proceed by constructing a graph, that represents these transitions for all partitions. The graphs SG and IG are defined as follows, for set and instance semantics respectively:

$$Nodes(SG) = Nodes(IG) = PCond$$

$$\forall c_1, c_2 \in PCond:$$
$$\exists o \in Q_R(db) \wedge c_1(o, db) \wedge c_2(o, Execute(R, db, set))$$
$$\Longrightarrow$$
$$(c_1, c_2) \in Edges(SG)$$

$$\exists o \in Q_R(db) \wedge c_1(o, db) \wedge c_2(o, Execute(R(o), db, instance))$$
$$\Longrightarrow$$
$$(c_1, c_2) \in Edges(IG)$$

Using this graph, we can reduce the problem of termination to the problem of cycle detection, which is a decidable problem. Confluence reduces to finding a unique sink from each node in the graph, which is also a decidable problem.

Left to be proven are Claim 6.1 regarding the decidability of consistency of a condition, and Claim 6.2 regarding the properties of partition conditions. □

Proof of Claim 6.1 We give an algorithm that checks a condition $\phi = \bigwedge_{i=1}^{n} \phi_i$ for consistency. Without loss of generality, we assume that the ϕ_i are ordered according to the following criteria:

1. Attribute-constant equalities before attribute-attribute equalities.
2. The attribute-constant equalities are sorted by attribute.
3. The attribute-attribute equalities are put in the form $a_i = a_j$ such that $i < j$ and then sorted lexicographically by the pairs (a_i, a_j).

The algorithm proceeds by constructing an object o with attributes $a_1, \ldots, a_n$, that satisfies the condition ϕ. For the construction, we need a set of dummy variables $Dummy$ with the following properties:

$$Dummy = \{D_1, \ldots, D_n\}$$
$$\text{such that} \quad (1)\ i \neq j \Rightarrow D_i \neq D_j$$
$$(2)\ a_i : \tau \Rightarrow D_i : \tau$$

The algorithm is given in Figure 9.3.

Successful termination of this algorithm means, that it was able to construct an object satisfying the given condition ϕ, implying consistency of ϕ. If the algorithm terminates unsuccessfully, then the condition is inconsistent. If the condition is consistent, we can construct a database state from the object and the set of dummy values.

The algorithm works in two phases. First (lines 3–10), conditions of equality of attributes to constants are checked. Constants are assigned to attributes for which such conditions exist. Other attributes are assigned dummy values. The

Algorithm consistency-check
Begin
Check $\forall a_i \in A$:
Case $\exists c_1, c_2 \in C_R : a_i = c_1 \wedge a_i = c_2$:
Exit(Unsuccessfully)
Case $\exists! c_1 \in C_R : a_i = c_1$:
$o.a_1 := c_1$
Otherwise:
$o.a_i := D_i$
Endcheck

Check $\forall \phi_k$ of the form $a_i = a_j$:
Case $o.a_i = c_1 \wedge o.a_j = c_2 \wedge c_1 \neq c_2$:
Exit(Unsuccessfully)
Case $o.a_i = c_1 \wedge o.a_j = c_2 \wedge c_1 = c_2$:
Next
Case $o.a_i = c_1 \wedge o.a_j = D_j$:
$a_j := c_1$
Case $o.a_i = D_i \wedge o.a_j = c_1$:
$a_i := c_1$
ReplaceAll(D_i, c_1)
Case $a_i = a_j \wedge o.a_i = D_i \wedge o.a_j = D_j$:
$a_j := D_i$
Endcheck

Exit(Successfully)

End.

Figure 9.3: Algorithm to check the consistency of a local condition

next part (lines 12–24) tries to satisfy equalities between attributes. Here, the order on the equalities prevents costly `ReplaceAll` actions for dummy variables, by completing assignment of dummies to lower-indexed attributes first.

To prove the correctness of the algorithm, we show that if it exits unsuccessfully, it has constructed an inconsistent state.

Line 5 In this case, the condition requires an attribute to have two different values. Clearly, this is inconsistent.

Line 14 In this case, the condition requires two attributes to be equal, while at the same time it requires equality of both attributes to two unequal constants. Clearly, this is inconsistent.

□

Proof of Claim 6.2 The first part of this claim was

$$\begin{array}{l}\forall db \in DB, \forall o \in db : \\ \quad \exists! pc \in PCond : pc(o, db)\end{array}$$

The existence of a $pc \in PCond$ is obvious from the fact that all possible equalities between attributes and constants are included in WEC, from which the partition conditions are constructed. The existence of a unique $pc \in PCond$ follows from the maximality of the partition conditions, because maximality of pc means $\nexists pc' \in PCond : pc' \Rightarrow pc$.

The second part of the claim was that

$$\begin{array}{l}\forall db_1, db_2 \in DB, \forall o_1 \in db_1, \forall o_2 \in db_2 : \\ \quad pc(o_1, db_1) \wedge pc(o_2, db_2) \\ \quad \Longrightarrow \\ \quad \exists pc' \in PCond : \\ \qquad pc'(o_1, Execute(R, db_1, set)) \\ \qquad \wedge \\ \qquad pc'(o_2, Execute(R, db_2, set))\end{array}$$

under set semantics and

$$\begin{array}{l}\forall db_1, db_2 \in DB, \forall o_1 \in db_1, \forall o_2 \in db_2 : \\ \quad pc(o_1, db_1) \wedge pc(o_2, db_2) \\ \quad \Longrightarrow \\ \quad \exists pc' \in PCond : \\ \qquad pc'(o_1, Execute(R(o_1), db_1, instance)) \\ \qquad \wedge \\ \qquad pc'(o_2, Execute(R(o_2), db_2, instance))\end{array}$$

under instance semantics.

This claim follows from the uniqueness of the partition conditions and the fact that the changes made by M_R are identical to both objects. □

The method used for deciding termination of a singleton trigger set can be extended to a trigger set with more than one trigger. The method used is the same, except that we must construct a graph using more than one trigger.

Theorem 7 *In the* DEGAS^{-2} *model, given a trigger set* $\mathcal{R}$ *and semantics* sem*, the predicates* $Terminate(\mathcal{R}, sem)$ *and* $Confluence(\mathcal{R}, sem)$ *are decidable.*

Proof Since the conditions and actions of the triggers are local to an object, we can treat this problem for each class separately. For each class C, we construct a set of partition conditions $PCond_C$ using the method used in the proof of Theorem 6. We then take the union of these sets into one set of partition conditions $PCond = \bigcup_C PCond_C$. Clearly, since each $PCond_C$ induces a typical database state tdb_C, their union $PCond$ also induces a typical database state $tdb = \bigcup_C tdb_C$, see [Voort, 1994, Chapter 5] for details.

After construction of the typical database state, we proceed with the construction of a graph. The presence of more than one trigger has some effect on the drawing of the graph. The graphs SG for set semantics and IG for instance semantics are defined as:

$$Nodes(SG) = Nodes(IG) = PCond$$

$$\begin{aligned}
&\forall c_1, c_2 \in PCond : \\
&\quad \exists R \in \mathcal{R}, \exists o \in Q_R(db) : c_1(o, db) \wedge c_2(o, Execute(R, db)) \\
&\quad \Longrightarrow \\
&\quad (c_1, c_2) \in Edges(SG)
\end{aligned}$$

$$\begin{aligned}
&\forall c_1, c_2 \in PCond : \\
&\quad \exists R \in \mathcal{R}, \exists o \in Q_R(db) : c_1(o, db) \wedge c_2(o, Execute(R(o), db)) \\
&\quad \Longrightarrow \\
&\quad (c_1, c_2) \in Edges(IG)
\end{aligned}$$

Again deciding termination reduces to cycle detection and deciding confluence reduces to finding a unique sink for each node. □

9.4 Decidability Results for DEGAS$^-$

In this section, we look into decidability of the predicates in DEGAS$^-$. The only difference between DEGAS$^-$ and DEGAS^{-2} is that DEGAS$^-$ allows path expressions in a rule's condition. This is sufficient to make termination an undecid-

able property. Some stronger properties, however, are decidable, such as termination in n steps and independence.

First, we consider termination of a singleton trigger set. This is a decidable predicate in DEGAS$^-$.

Theorem 8 *In* DEGAS$^-$*, given a singleton rule set* $\mathcal{R}$ *and semantics* sem *the predicate* $Terminate(\mathcal{R}, sem)$ *is decidable.*

Proof We show that we can construct a typical database state in this model. A complication is, that the number of possible conditions is infinite.

Since path expressions are allowed in the condition of a rule, the set of elementary conditions EC is generated by the following grammar:

$$\begin{aligned} \langle \text{Econd} \rangle &\longrightarrow \langle \text{AttrExpr} \rangle = \langle \text{BasicExpr} \rangle \\ \langle \text{BasicExpr} \rangle &\longrightarrow \langle \text{AttrExpr} \rangle \mid \langle \text{Const} \rangle \\ \langle \text{AttrExpr} \rangle &\longrightarrow \langle \text{Attr} \rangle \mid \langle \text{Attr} \rangle . \langle \text{AttrExpr} \rangle \end{aligned}$$

where the non-terminal ⟨Attr⟩ yields all attributes in the schema. ⟨Const⟩ yields all constants used in the rule set. We use the obvious typing rules to restrict EC to the set of well-typed elementary conditions WEC. We collect all possible conditions with conjunction and disjunction into the set $Cond$.

Because of the complexity of conditions with path expressions, we introduce the length of a condition. Intuitively, this is a measure of the distance of the attributes of interest to the condition from the local object. The length of a condition is recursively defined as:

1. $length(e) = 1$, where e is an attribute identifier or a constant.
2. $length(a.e) = length(e) + 1$, where a is an attribute and e is a path expression.
3. $length(\bigwedge_i \bigvee_j (e \omega f)_{ij}) = max(\{max(length(e), length(f))\}_{ij})$, where e is a path expression and f a path expression or a constant.

$Cond_n$ denotes the set of conditions of maximum length n.

Consistency of a condition can be checked by a slight modification of the algorithm in the proof of Theorem 6. The difference in the conditions is the possibility to refer to other objects. Hence, in order to show that a database satisfying the condition exists, we need to construct more than one object. In fact, we construct all objects referred to in the condition. Using this consistency check and a partial order as defined previously, we obtain at the set of partition condition of length n:

$$PCond_n = \{c \in Cond_n | c \text{ is consistent} \wedge c \text{ is maximal wrt } >\}$$

As before, this set induces a typical database state tdb_n, that can be constructed using the consistency checking algorithm.

Now, we can again construct a graph to encode the execution of a rule. A complication is that an object can move from one partition condition to another, because of a change in another object. Therefore, we label the transition to indicate, whether it is a direct or an indirect transition. A direct transition, labelled da, is caused by the direct application of a rule to the object. An indirect transition, labelled in, is caused by the execution of a rule on another object.

We construct a graph for instance semantics $IG_{n,\mathcal{R}}$ and set semantics $SG_{n,\mathcal{R}}$, separately.

1. $\forall n \in \boldsymbol{N} : Nodes(SG_{n,\mathcal{R}}) = Nodes(IG_{n,\mathcal{R}}) = PCond_n$

2. $\forall n \in \boldsymbol{N}, \forall c_1, c_2 \in PCond_n :$
   $\exists o \in Q_R(tdb_n) :$
   $c_1(o, tdb_n) \wedge c_2(o, Execute(R, tdb_n, set))$
   $\Longrightarrow$
   $(c_1, c_2, da) \in Edges(SG_{n,\mathcal{R}})$

   $\exists o \in Q_R(tdb_n) :$
   $c_1(o, tdb_n) \wedge c_2(o, Execute(R(o), tdb_n, instance))$
   $\Longrightarrow$
   $(c_1, c_2, da) \in Edges(IG_{n,\mathcal{R}})$

3. $\forall n \in \boldsymbol{N}, \forall c_1, c_2 \in PCond_n :$
   $\exists o \notin Q_R(tdb_n) :$
   $c_1(o, tdb_n) \wedge c_2(o, Execute(R, tdb_n, set))$
   $\Longrightarrow$
   $(c_1, c_2, in) \in Edges(SG_{n,\mathcal{R}})$

   $\exists o_1 \in Q_R(tdb_n) :$
   $c_1(o, tdb_n)$
   $\wedge$
   $\exists o_2 \in tdb_n :$
   $o_2 \in Q_R(tdb_n) \wedge o_1 \neq o_2$
   $\wedge$
   $c_2(o_1, Execute(R(o_2), tdb_n, instance)$
   $\Longrightarrow$
   $(c_1, c_2, in) \in Edges(IG_{n,\mathcal{T}})$

This graph encodes the evolution of an object under the execution of the singleton rule set $\mathcal{R} = \{R\}$, if we take $n = length(C_R)$. This graph has the useful property, that:

Claim 8.1 *In the graphs $SG_{n,\mathcal{R}}$ and $IG_{n,\mathcal{R}}$, there are no cycles of a length greater than* 1.

Thus, deciding termination amounts to the detection of da-cycles of length 1, since this indicates the possibility of infinite application of a trigger to an object. This is a decidable problem. Confluence of a trigger set can be decided by checking whether each object has a unique sink. □

Proof of Claim 8.1 The first thing to be noted, is that we have taken the value of n, such that any change seen by an object through the trigger condition is included in the partition condition. In addition, the use of disjunction in the partition conditions means that other objects' states that are indifferent to the local object are included in the partition condition. We should also keep in mind, that application of the action to an object is idempotent.

There are three possible configurations for cycles of length greater than 1:

1. Cycle consisting of da-edges only
2. Cycle consisting of in-edges only
3. Cycle consisting of at least one da-edge with the other edges in-edges.

Because of the idempotence of rule application, cycles consisting only of da-edges must be of length 1. A cycle consisting of more than one da-edge would mean, that the local state of the object changes after the first application of the rule to that object. This is in contradiction to the idempotence of rule application in DEGAS$^-$.

The value of n is chosen in such a way, that all variables of importance to the object are incorporated in the partition conditions. Since there is no change in the local state during an in-transition, there must be another object that changes. However, the local state of any object changes at most once during any rule execution. Therefore, it is not possible that an object's state returns to its initial state after the first rule application to it. This also implies the impossibility that all objects referring to a partition condition return to their initial states. This is exactly what would happen, if an in-cycle were present in the graph. Therefore, a cycle of more than one in-edge cannot exist.

For the next case, we first show that we need only consider mixed da-in-cycles with one da-edge. Suppose a mixed cycle contains more than one da-edge, one from c_1 to c_2 and the second from c_i to c_{i+1}. The local state of the object does not change with the second application of the rule. Therefore, there must be a da-edge from c_1 to c_{i+1}. This means that there always is a shorter cycle, if there are more than one da-edge.

Now, we consider a cycle consisting of one da-edge from c_1 to c_2 and of in-edges otherwise. The argument for the non-existence of such a cycle is the

same as for the non-existence of a cycle of only *in*-edges. This is also obvious, because the existence of such a mixed cycle requires that the object itself reverts to its initial state.

Thus, every cycle in the graph must be of length 1. As a result of this, we can construct a database for each path through the graph. It is obvious that a database can be constructed for a single transition in the graph. The method used is the consistency checking algorithm mentioned earlier. This can be extended in a straightforward way for an acyclic graph. It is also obvious that we can construct a database that follows a cycle of length 1. □

Although termination is decidable for a singleton trigger set, it is not for a set of more than one trigger. The reason is, that the DEGAS$^-$ model is powerful enough to simulate a Turing Machine. In a Turing Machine, all replacements of values on the tape is by constant symbols. Only in moving the head, we need communication between cells, which can be done by reading a status attribute at the neighbouring cell.

We first show, how a Turing Machine is emulated using DEGAS$^-$ objects. The specification of a cell, without the rules, is given in Figure 9.4. Each cell on the tape is represented by an object. It contains the identities of its neighbouring cells in the attributes `left-neighbour` and `right-neighbour`. The attribute `value` contains the symbol on the tape in this cell. The state of the Turing Machine is recorded in the attribute `state`, if the head is on this cell. The attribute `current` indicates whether the head is on this cell. The other two attributes, `next` and `from`, are used during movement of the head.

In this attribute specification, the types Symbol, State and all finite sets can be considered subsets of Integer, with the constants denoting a certain number. No extra functionality is added to DEGAS$^-$ by using these types.

The transition table is recorded in the rules that each record one entry of the transition table. These rules are activated, when a cell becomes the current cell. To make sure the right cell is designated as the current cell, we need a number of bookkeeping rules, two each for moving the head to the left and the right. Please note, that we have labelled the rules for ease of discussion.

The execution of a transition is triggered on a cell, when the cell is current and the previous cell is finished. Thus, the execution rule for the transition determined by the state s and value v becomes:

*/*** Execute(s,v) ***/*
If current=yes $\wedge$ from.next=neutral $\wedge$ from.state=s $\wedge$ value=v
do execute($x_1 = c_1, x_2 = c_2, x_3 = c_3$)

where c_1 denotes the new symbol for the cell, c_2 the direction the head moves to and c_3 the new state of the Turing Machine.

```
Object Cell
Attributes
   left-neighbour : Cell
   right-neighbour : Cell
   value : Symbol
   state : State
   next : { left, right, neutral }
   current : { yes, no }
   from : Cell
Methods
   execute(x1:Symbol, x2:{left,right}, x3:State) = {
      value:=x1
      next:=x2
      state:=x3
   }
   become-current-left() = {
      from := right-neighbour
      current:=yes
   }
   become-current-right() = {
      from := left-neighbour
      current:=yes
   }
   not-current-anymore() = {
      current:=no
      next:=neutral
   }
Rules
   ...
EndObject
```

Figure 9.4: Specification of a cell in the DEGAS⁻ Turing Machine emulation

After the transition is executed, the head must be moved to the next cell. If the head moves to the left, the cell that is next, must change its status to current. It knows it may do this, if its right neighbour has recorded that the head moves left. The action to be taken is the method `become-current-left` This is specified by the following rule:

*/*** To-be-current-left ***/*
If right-neighbour.next=left $\wedge$ right-neighbour.current=yes
do become-current-left

When the destination cell has registered that it is the current cell, the previous cell must set `current` to *no* and erase the movement data in `next`.

*/*** Was-current-left ***/*
If next=left $\wedge$ current=yes $\wedge$ left-neighbour.current=yes
do not-current-anymore

The latter two rules are also defined for a head movement to the right. Their definition is analogous to the rules for the left movement, with the appropriate substitutions of left and right.

To give a better insight in the emulation of a Turing machine by these rules, we show how these rules achieve the movement of the head to the left. The two `cell` objects of interest are shown with their contents. We start right after the application of an `Execute` rule on the right cell. At that time, the attribute valuations are as follows:

	Oid=*Newcell*	Oid=*Oldcell*	
	current=no	current=yes	
	next=neutral	next=left	
	from=?	from=?	
	state=?	state=s	
	value=v_1	value=v_2	
	right-neighbour=*Oldcell*	right-neighbour=?	
	left-neighbour=?	left-neighbour=*Newcell*	

Object *Newcell* satisfies the condition of `to-be-current-left`, so the method `become-current-left` is executed on this object. This results in the following attribute valuations:

	Oid=*Newcell*	Oid=*Oldcell*	
	current=yes	current=yes	
	next=neutral	next=left	
	from=*Oldcell*	from=?	
	state=?	state=s	
	value=v_1	value=v_2	
	right-neighbour=*Oldcell*	right-neighbour=?	
	left-neighbour=?	left-neighbour=*Newcell*	

Now, *Newcell*'s `current` attribute has been set. Hence, `current` must be set to *no* in *Oldcell*. This is done by the rule `Was-current-left`, that executes on *Oldcell*. The result is:

	Oid=*Newcell*	Oid=*Oldcell*	
	current=yes next=neutral from=*Oldcell* state=? value=v_1 right-neighbour=*Oldcell* left-neighbour=?	current=no next=neutral from=? state=s value=v_2 right-neighbour=? left-neighbour=*Newcell*	

Now, the condition for the rule `Execute(s,`v_1`)` is satisfied and its action is executed. After that, the head is again moved by the bookkeeping triggers.

A final note on the emulation of a Turing Machine by a DEGAS⁻ database is on starting the Turing Machine. The `Execute(s,v)` rules are triggered by the neutral state of the previous cell in the execution. However, there is no previous cell at the start. Therefore, we need an extra cell, that is not part of the tape and has its `next` attribute set to neutral. The `from` attribute of the starting cell, i.e., the cell where the head is positioned at the start of the execution, is set to this extra cell.

It is obvious that the given bookkeeping rules correctly implement the head movement. For each entry in the transition table of the Turing Machine, we can define an `Execute` rule. A transition consists of a tape symbol and a state of the machine, resulting in writing a new value on the tape, moving the head left or right and a new state. These can be translated to triggers by filling in these values for s, v, c_1, c_2 and c_3, respectively.

In order to show that this emulation of a Turing Machine is the same under instance and set semantics, we have to show that no rule is executed on more than one object at the same time. The attributes governing rule application are `current` and `next`. If we start with a correct input state, only one `cell` object will have `current` set to *yes* and all `next` attributes will be set to *neutral*. Obviously, as long as only one object has `current` set to *yes*, any rule is only executed on one object. The only time `current` equals *yes* in two objects simultaneously is during the movement of the head. This, however, immediately triggers the `Was-current-{left,right}` rule, that sets the `current` attribute of the previous cell to *no*. No other rule is triggered in this situation.

Lemma 1 *For each Turing Machine TM exists a pair* $(\mathcal{R}, db)$ *with* $\mathcal{R}$ *a rule set and* db *a database state, such that* $(\mathcal{R}, db)$ *implements TM under both instance and set semantics.*

The possible emulation a Turing Machine in DEGAS⁻ gives a clear indication of the decidability of termination of a rule set. The halting problem for Turing Machines is known to be undecidable. Therefore, termination of a set of rules in DEGAS⁻ is also an undecidable problem.

Theorem 9 *Let* $\mathcal{R}$ *be a trigger set in* DEGAS⁻ *and let* sem *denote either instance or set semantics.* $Terminate(\mathcal{R}, sem)$ *and* $Confluent(\mathcal{R}, sem)$ *are undecidable predicates in this case.*

Proof According to Lemma 1, everything that can be computed on a Turing machine, can be computed by using DEGAS$^-$ rules. Suppose we could decide termination of a set of rules in DEGAS$^-$. Then, we could solve the problem whether a Turing Machines terminates on every input by translating the Turing Machine to a DEGAS$^-$ database. Termination, however, is undecidable for Turing Machines (see, e.g., [Lewis and Papadimitriou, 1981]). Therefore, termination of a set of rules in this model must be undecidable.

Since

$$Confluent(\mathcal{R}, sem) \Rightarrow Terminate(\mathcal{R}, sem)$$

and $Terminate(\mathcal{R}, sem)$ is undecidable, $Confluent(\mathcal{R}, sem)$ is also undecidable. □

The possibility of simulating a Turing Machine points us to a stronger predicate, that is decidable for Turing Machines, viz., termination of a set of rules in n steps. The restriction to a limited number of steps imposes an upper bound on the time a decision procedure can take. We can simply run the rule set on a typical database state of sufficient length, until we reach the maximum number of steps. We then check whether rule execution has terminated or not.

A similar strategy is applied to solve a stronger predicate than confluence, pairwise independence of rules. This means that the both possible sequences for two rules yield the same database state. If all rules are pairwise independent, and the rule set is terminating, then the rule set is confluent. We can test this on the typical database state by running both possible executions of each pair and comparing the result. Thus, we have the following theorem:

Theorem 10 *For a rule set $\mathcal{R}$, semantics sem and an integer number $n > 0$ the predicates $Terminate(n, \mathcal{R}, sem)$ and $Independent(\mathcal{R}, sem)$ are decidable* DEGAS$^-$.

As a consequence, termination and confluence are decidable for independent rule sets.

Corollary 1 *$Terminate(\mathcal{R}, sem)$ and $Confluent(\mathcal{R}, sem)$ are decidable predicates for an independent rule set $\mathcal{R}$ and semantics sem.*

Proof If all rules in $\mathcal{R}$ are independent, we can rearrange an execution sequence of $\mathcal{R}$ at will. Let $\mathcal{R} = \{R_1, \ldots, R_n\}$. Any execution sequence of $\mathcal{R}$ can be rearranged into the form $R_1; \ldots; R_1; R_2; \ldots; R_2; \ldots; R_n; \ldots; R_n\}$. For each $R_i \in \mathcal{R}$ individually, we can decide termination. If each $R_i \in \mathcal{R}$ terminates individually, then it is obvious that $\mathcal{R}$ terminates.

Because we can rearrange an execution sequence of an independent trigger set $\mathcal{R}$ at will, $\mathcal{R}$ is obviously confluent. □

9.5 Conclusion

In this chapter, we have explored the decidability of active rules in two restricted version of DEGAS, DEGAS$^-$ and DEGAS^{-2}. In the simplest language, DEGAS^{-2}, termination and confluence are decidable properties. The mere addition of path expression in the selection condition, in DEGAS$^-$, makes these properties already undecidable. Additionally, the emulation of a Turing machine by a very simple rule language shows the power of the active rule paradigm. Since an event-condition-action language is a superset of a condition-action language, the undecidability results for termination and confluence also apply to ECA rules.

A consequence of these undecidability results for DEGAS$^-$, termination and confluence are also undecidable for the full DEGAS model. In fact, termination of a single DEGAS rule is already undecidable, because we can translate the Turing Machine emulation of Section 9.4 to a single DEGAS rule and method using the `forall...in...where...do` construct.

Most practically useful active database systems need to incorporate a more complex rule language than those considered in this chapter. Therefore, we need to find other ways to obtain the desired properties of trigger sets. One of the approaches to guarantee termination and confluence are sufficient conditions. It is possible to formulate conditions on rule sets, that guaranteed termination and confluence of the set. Not all terminating rule sets, however, satisfy a sufficient condition for termination. Thus, a number of terminating rule sets will be rejected for use. Alternatively, we can take a purely empirical path and monitor the active database system in use. If the number of successive rule applications exceeds a preset limit, we cancel rule execution. This prevents a system from getting stuck in rule execution, but it does nothing to prevent the re-occurrence of the non-terminating rule application. Another problem of monitoring is, that we cannot monitor a property like confluence. To detect non-confluence, rule application would have to be repeated twice in exactly the same database state.

To overcome the drawbacks of both mentioned approaches, we need a more sophisticated method. We propose to build a learning capability into the active objects. Thus, an object is able to learn to avoid the situation where it gets into, for example, a non-terminating rule application. In order to avoid such situations, an active object will be able to change its rules. This requires a different specification of the active object's functionality, possibly borrowing from the area of intelligent agents as discussed in Chapter 10. This kind of strategy, however, does not solve the problem of confluence.

Part IV

Outlook and Conclusions

Chapter 10

Outlook

Previous chapters of this thesis introduced and formalised the DEGAS model for a database of autonomous objects. Furthermore, we discussed the design and verification of DEGAS database. This chapter takes a brief look at future directions of computing environments based on autonomous objects. Here, we assume the move towards increased distribution of computing already mentioned in Section 2.1. A model for distributed computing that has generated much interest in the research community is the notion of intelligent agents [Shoham, 1993, CACM, 1994]. Hence, we combine databases and agents in two ways in this chapter. First, we examine the opportunities of agent technology in active databases. Second, we look at a data management environment based on agents.

10.1 Active Databases and Agents

The ongoing miniaturisation of computers is leading us to a world, where computers are omnipresent. This phenomenon has become widely known under the name *ubiquitous computing* [Weiser, 1993, Abowd, 1996]. Although this development is still in its early stages, there is already some consensus on the software architecture for ubiquitous computing. This consensus considers *intelligent agents* [Shoham, 1993, CACM, 1994] the most promising paradigm for future information systems. In this paradigm, software consists of a number of entities collaborating towards a common goal, functioning autonomously with little intervention. Hence, it is reasonable to expect that, in future computing environments, information systems will be based on a large number of cooperating agents.

At the same time, research in databases, the traditional foundation of an information system, has addressed the inclusion of additional modelling notions in databases. This has lead, among other things, to active databases, i.e., databases

that include production rules. This allows databases to react autonomously to certain situations in the database.

Since databases are the current foundation and agents a future foundation of information systems, the question rises how information systems might evolve from databases to agents. Since active databases are the first step in this evolution, we examine the role agent technology can play in an active database. The possibility of integrating agents in active databases was earlier mentioned in [Bailey *et al.*, 1995], which compared active databases and agent systems. Its main focus, however, was on the similarity of concepts in both areas, whereas we look into the opportunities agent technology offers active databases.

In this section, we first identify the "level of agency" in DEGAS. Then, we present our view on the benefits the addition of further features of agent technology can bring to active databases.

10.1.1 Agency in Active databases

Research on agents generally distinguishes *weak* and *strong* agency. A software system is said to have weak agency, if it possesses the following four properties [Wooldridge and Jenning, 1995]:

- autonomy
- social ability
- reactivity
- pro-activeness

Object *autonomy* is one of the base assumptions in DEGAS. Each DEGAS object is itself a process. Furthermore, its dependence on other objects is as small as possible through complete encapsulation and minimal assumptions about the behaviour of other objects. Hence, the criterion of autonomy is satisfied by DEGAS objects. *Social ability* means that agents interact with other agents in the system through an agent communication language. DEGAS objects pass messages to other objects and engage in relations with them. DEGAS objects *react* to their environment by answering messages. Furthermore, rules also specify reactions to situations that occur in the DEGAS database. *Pro-activeness* means that agents can take the initiative to achieve certain goals. Although goals are not explicit in a DEGAS object, active rules are instrumental to achieving a goal.

There is less consensus over stronger levels of agency. In general, strong agency is concerned with mentalistic notions. For the discussion in this paper, we take the four dimensions formulated by Shoham [Shoham, 1993]:

- knowledge
- belief

- intention
- obligation

These notions are not explicitly supported by DEGAS objects. Although rules can be used to express obligations, they are not formulated as such. An obligation of an agent is specified by a goal that must be achieved. Instead, an ECA rule is just an instruction to execute a certain action in a specified situation, although this action will be instrumental in fulfilling the obligation.

Likewise, intention is only implicitly present and, as far as it is present, not arrived at by the DEGAS object itself. We could say, that a rule to maintain a database constraint expresses the intention to maintain that constraint. Intention and instrument to realise this intention, however, are fixed to each other. If intention is an independent notion to an agent, it first derives its intention and then reasons about the actions to realise it.

Knowledge and belief are unknown notions in an active database. Although a lot of information is stored, the way to process these data is fixed by the methods and rules specified. Furthermore, a database usually lacks the ability to reason with and about the information it contains in a general way.

10.1.2 Extending Agency in an Active Database

In the previous section, we saw that DEGAS supports weak agency. In addition, limited representation of obligation and intention are present. In this section, we look at the potential results of extending the level of agency in an active database, taking DEGAS as a starting point. In particular, we consider the benefits of stronger agency for general database functionality.

Stronger agency is introduced in an active database by extending the capabilities of the objects in the database. While DEGAS currently is a database of autonomous objects, we would then have a database of agents. The agents in such a database[1] each manage a part of real world data, like an object represents a piece of data. This means that an agent contains a piece of data, and additionally possesses a number of goals it has to achieve or maintain. Furthermore, a data agent will have a number of obligations. In part, these will exist to facilitate DBMS functionality, e.g., an obligation to answer queries. Another part of the obligations will be to other agents, caused by relations between agents.

The key advantage of the promotion of autonomous objects to agents lies in the reasoning ability of agents. This allows a more abstract specification of the database, both in application modelling and in implementing database functionality. Triggers implement a tight binding between goals and means, so that

[1] or would we have to call it a data management society?

an object has only one means to achieve a specific goal. By formulating goals and means separately, more flexible solutions are possible for information system functionality.

A prime example of the additional flexibility provided by separate goals and means is given by constraints. In Chapter 3, we mentioned that triggers were originally devised to deal with constraints more flexibly. The improvement triggers offered over existing mechanisms was, that we could use different strategies to maintain different constraints, instead of a single strategy for all constraints. Strong agency, by separating goals and means, gives us additional flexibility by allowing multiple strategies to maintain a single constraint. The agent can then infer which strategy is optimal in the current situation. In addition, the presence of general problem-solving strategies in the database obviates the need for specialised compilers, e.g., to produce rules to maintain constraints [Ceri and Widom, 1990].

As an example consider the limit on the negative balance of a bank account. A limit of 2000 in the red is specified by the constraint:

$$balance \geq -2000$$

Suppose now that a requested payment violates this constraint. In this situation, we have a number of strategies to enforce this constraint, of which we mention four:

1. Refuse the payment
2. Transfer funds from a savings account
3. Sell some shares
4. Arrange a temporary loan

If we were to use triggers, we can specify only one strategy to enforce the limit on this account. With its enhanced reasoning capabilities, an agent can choose the best strategy to enforce this constraint, given its other goals, such as the quality of its relation with the customer, income in the near future etc.

Another advantage of stronger agency over triggers, is found in the problem of deciding termination of trigger sets. As we saw in Chapter 9, this is decidable only for very simple rule languages. Hence, we must find another way of dealing with this problem than deciding it in advance or imposing conservative pre-conditions on trigger sets. Stronger agency can help counter the problem of non-termination in two ways. First, the separation of multiple means to achieve the goal of a trigger, allows an agent to choose another means to achieve its goal, if the means originally chosen has undesirable side-effects. Second, not every possible execution of a non-terminating trigger set is non-terminating. This means, that the agents can cooperate to avoid the non-terminating execution sequence of their triggered actions.

As an example, consider the trigger activation graph given in Figure 10.1. In this graph, the nodes are database states and the edges are trigger executions. For example, in database state c trigger $t3$ is triggered, whose execution brings the database in state e. The trigger set in this graph is non-terminating, since there exists a cycle of triggers $t2$ and $t1$ in this activation graph, which keeps the database shuttling between database states b and c. We can easily see, however, that there is also a terminating execution sequence for this trigger set, viz., the sequence $t1;t3;t2;t1$ that leads to the stable state f.

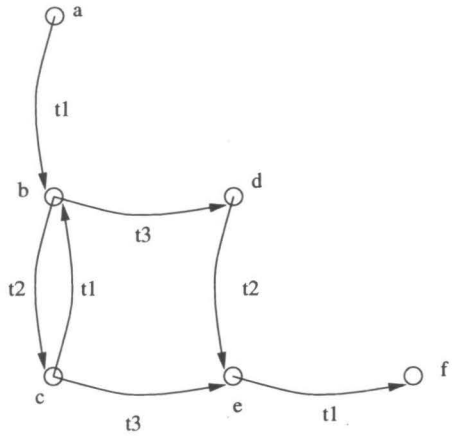


Figure 10.1: A trigger activation graph

The advantages given above of strong agency over the weak agency in an active database also apply to dynamic database constraints. In DEGAS, the dynamic constraints of an autonomous object are given by the lifecycles. This lifecycle is fixed. Hence, a message that does not fit in the lifecycle is rejected outright. If an autonomous object had higher level reasoning facilities, it would be possible to negotiate a deal with the sending agent. For example, the receiving object might give an indication of the time when it will be able to execute the requested action. The sending object can then decide, depending on its other goals and obligations, whether it can wait or take another course to achieve its goal.

If negotiations between agents are to take place between different agents in the databases, we need, besides a language to conduct the negotiations in, a measure of the value of the different propositions being negotiated. This implies the use of a monetary model. The use of such a model is experimented in a somewhat different context in the Mariposa system [Stonebraker *et al.*, 1996]. Here, the allocation of data storage and query processing in a distributed database is managed through a bidding system. The fixed bidding protocol in Mariposa, however, leads to undesirable effects in the allocation of data. In

fact, it turns out that the richest site will end up with all the data, which means that it only gets richer by consequently under-bidding the other sites for query processing. Clearly, more sophisticated bidding and negotiation protocols are needed, which leaves the components (agents) in the database more room for manoeuvring to avoid undesired outcomes. For example, in the example of the richest site taking all data, the other sites might temporarily adopt a "dumping" strategy, i.e., working at a loss in order to gain back work and data.

The examples given of agents working out solutions for databases problems by negotiating between themselves, can only work under certain assumptions about their intentions. As research in game theory has shown [Rosenschein and Zlotkin, 1994], it is difficult to come up with strategies without undesirable outcomes, if not all players are cooperative. Therefore, agent research often assumes, as [Shoham, 1993] does, the veracity and benevolence of agents towards each other. This assumption cannot be held if agents in our information system must interact with agents owned by other people or organisations. Hence, agent research must find a solution for dealing with uncooperative, lying and malevolent agents in order to be able to form the foundation of an information system, or an information infrastructure in general.

Further potential of agent technology in databases is found on the architectural side. If agent technology matures enough to form the basis of a database management system, this will implicate a large gain in flexibility of a DBMS. The different components of a DBMS, such as query optimiser, storage manager, etc., can each be an agent with its own goals and strategies. Besides the increased flexibility of the individual components, this also means increased freedom for a systems designer to choose the components constituting his DBMS.

10.1.3 Conclusions

In this section, we discussed the autonomous behaviour that active databases add to traditional databases. We saw that DEGAS implements weak agency, since DEGAS objects interact with each other, react to their environment, and autonomously pursue their defined goals. These functions, however, are limited by their fixed, pre-programmed character.

The extension of autonomous DEGAS objects to stronger agents with general purpose reasoning abilities greatly increases the adaptability and flexibility of an active database system. The improvements originate in the ability to adapt strategies to the actual situation, and the ability of agents to cooperate with each other. Hence, the incorporation of agent technology in databases opens up new perspective in tackling long-standing database issues. Increased coupling and inter-operation of information systems, however, also poses some new challenges for agent technology in order to deal with lying or malicious agents from outside.

10.2 Ubiquitous Databases

As already mentioned in Section 10.1, we see a move towards ubiquitous computing [Weiser, 1993, Abowd, 1996]. In a ubiquitous computing environment, computers are everywhere and blend seamlessly into the physical environment. This kind of computing environment is stimulated by the ever decreasing cost of processing power and memory. This section discusses the impact of ubiquitous computing on data management.

As a general rule, information recorded by computers is data previously recorded on paper. Initially, computers were used to store large paper file collections in centralised databases, mainly in administrative applications. Nowadays, there is hardly an organisation recording and processing large volumes of data that does not use computers. Relational database management systems are particularly suited for managing large volumes of simply structured data.

From applications with bulk data, the use of computerised data storage spread to applications with more complex data. Additionally, the diversity in structure of the data is much higher. An example are computer-aided design systems. The elaborate modelling concepts in object-oriented databases make them highly suitable to manage the complexly structured data in this type of application.

As a next step, the World Wide Web, including intranets and extranets, is a prime example of the increasing use of computers to store less-structured data. Information previously distributed and stored on paper is nowadays distributed and stored electronically. Furthermore, almost every piece of information in this environment has a different structure, meaning a further increase in diversity of data stored.

Extrapolating the development of data storage, all written information will ultimately be stored in electronic form. Furthermore, we see an increasing popularity of mobile computing and communication devices. As a result of these developments, we will see an increasing demand for information to be available in computerised form everywhere. In such a situation, the model of a centralised database is not tenable anymore. The low cost and widespread distribution of computing power means that the best place to store data on artifacts in the real world will be these artifacts themselves.

An application area where a move towards ubiquitous computing might be beneficial is the shipment of maritime containers. Currently, the physical flow of goods and the information flow are separated. For example, the information needed to compile the bill of lading, describing a ship's cargo, is taken from the transport orders of each container. In a paper-based system, the transport order is sent to the container terminal in advance of the container's arrival by train or lorry at the terminal. Although this might be handled by an Electronic

Data Interchange system, the actual arrival of the container still needs to confirmed manually.

Suppose now, that each container has its own small computer with a wireless networking capability. All necessary information about the container and its load are recorded on this computer. Then, a ship's bill of lading is generated simply by aggregating these data from all containers on the ship. The moment a container is loaded onto the ship, it is added to the bill of lading. Furthermore, this functionality increases the efficiency of the terminal. If a lorry loaded with a container arrives at the port, the driver no longer needs to hand over the container manually. Instead, the container's arrival is registered automatically at the gate and the required information is transferred to the terminal management system. The driver can then immediately be directed to the place to unload the lorry.

What are the requirements on a model for a ubiquitous computing environment? The assumptions underlying DEGAS we gave in Section 2.4 also apply to the objects in a ubiquitous computing environment:

1. Every object has a separate thread of execution.
2. Complete encapsulation of the behaviour of an object.
3. Strictly regulated access to an object.
4. Minimal guarantees about an object's behaviour to other objects.
5. Minimal dependency of an object on the behaviour of other objects.
6. Autonomy must be given up explicitly.

As a result of these assumptions, DEGAS object function autonomously on an infrastructure that facilitates object creation and communication. Two main modifications need to be made to make the DEGAS model fit for ubiquitous data management. First, objects need to be aware of their location. Second, the class structure must be changed to conformance-based model.

Addition of location awareness happens both at the hardware and at the software side of a computing unit. A computing unit must have hardware to determine its position. This may be based on a cellular system or on a GPS-based[2] device. On the software side, every object has a standard attribute `location` recording its location. This attribute is fed by the hardware part of the computing unit. Location information can then be used to establish relations based on nearness. For example, a container has a relation with the ship it is on, which is established at the time the container is loaded onto the ship. In a cellular system an object always has a relation with the cell it is in. As part of this relation,

[2]Global Positioning System. See [Hofmann-Wellenhof *et al.*, 1994] for details.

the cell can inform other objects of the object's presence. For example, the cell representing the container terminal's area informs the terminal management system of new container objects entering the cell. This kind of functionality is well-suited to be modelled by ECA rules.

To cater for the diversity of data in a ubiquitous computing environment, the class structure of DEGAS needs to be loosened. Instead of creating objects as instances of a class, classes become classifications of objects. An object belongs to a class, if it has the capabilities defined by the class. It can, however, have additional capabilities not present in other instances in the class. This kind of object classification was proposed in, among others, the Goblin database programming language [Kersten, 1991]. In such a system, objects can be created by cloning. Furthermore, it means a generalisation of the DEGAS addon mechanism to arbitrary additions of individual capabilities.

10.3 Conclusion

In this chapter, we provided an outlook on future evolution of DEGAS. The key notion in this outlook is agency. In Section 10.1, we discussed the impact of agents on active databases. We showed that an active object-oriented database, like DEGAS, already implements weak agency. Furthermore, stronger agency in an active database will allow increased flexibility in dealing with application constraints. It also offers new opportunities to solve long-standing issues in active databases.

Section 10.2 took the reverse view. It discussed data management in ubiquitous computing environments based on agents. From this discussion, we learned that incremental modifications of the DEGAS model yield a model for a ubiquitous computing environment, viz., addition of location awareness and a looser class structure.

Chapter 11

Conclusion

In this thesis, we formulated DEGAS, a database of autonomous objects, which positively answered the research questions in the problem statement in Chapter 1. The formulation and application of DEGAS led to the following answers:

1. A DEGAS database is easy implementable, since an implementation requires only two entities, a basic DEGAS object and a system layer, as was discussed in Chapter 7.
2. DEGAS facilitates a clean, modularised application design, as was shown in Chapter 8. The application designs are characterised by small units of functionality, that facilitates easy understanding of the design.
3. DEGAS has a direct, straightforward formalisation (given in Chapter 5), that has the additional advantage of formalising a historical object database.

Furthermore, DEGAS contributes to research in active, temporal, and object-oriented databases in a number of ways. The main contributions of DEGAS to database research can be summed up as follows, in order of importance:

1. **Modularisation of an active database.** A wider application of active rules leads to large rule sets. To manage these rule sets some form of modularisation is needed. In DEGAS, rules are encapsulated in objects. Furthermore, encapsulation of rules is a consequent application of object-oriented principles to active databases. Chapter 8 showed how this encapsulation facilitates a clear application design, showing the quality of the DEGAS modelling primitives.
2. **Object evolution.** DEGAS' addon mechanism provides a straightforward mechanism for object evolution. This mechanism is well-suited to implement object roles. Furthermore, the combination of the addon mechanism with rule encapsulation facilitates a clear modularisation with just-in-time availability of capabilities, as was shown through the workflow example in Chapter 8.

3. **A process-algebraic, formalisation and integration of active and historical databases.** The formalisation of DEGAS semantics by process algebra has two main advantages. First, it shows that active and historical databases can be specified by a single semantics. In particular, active rule semantics are defined relative to an object's history. Second, the use of process algebra for event expressions allows a direct definition of the semantics without intermediate translations.

4. **Queries in an active database.** The event detection facility to support rules in an active database can also be used to specify temporal conditions. Hence, the defined query model for DEGAS has an event-condition combination as the selector of a query. This gives us novel ways to specify temporal conditions. An added advantage is that the semantics of rules and queries share the event-condition part.

5. **Undecidability results for rules.** We showed that termination and confluence are undecidable predicates for rule sets in a subset of DEGAS. Hence, these are predicates are also undecidable for DEGAS

The development of DEGAS learned us that a database of autonomous objects is a feasible proposition in a number of aspects. The formalisation of a database at an object level shows clear advantages, especially in the integration of active and historical dimensions. The straightforward implementation of a DEGAS database based on just two primitive elements, the basic DEGAS object and a system layer, contributes to the practical attractiveness of DEGAS. From the design of a workflow application we learned that the DEGAS model leads to easy-to-understand and flexible object designs for applications.

Furthermore, DEGAS offers its innovations in a model that is an evolution of existing object models. A standard object design using only attributes and methods can be translated to DEGAS without any modification besides the addition of a lifecycle. Hence, DEGAS facilitates a gradual migration path from traditional object models.

A number of DEGAS' aspects offer perspective for further research. These concern the formalisation of object semantics, the facilities for evolution, and the looser coupling of capabilities and objects.

One of the innovative aspects of DEGAS is the integrated formalisation of active and historical databases. In DEGAS, this combination arose from a careful analysis of an active database's requirements. A generalisation of this model to temporal databases with multiple temporal dimensions raises some interesting issues. These are mainly related to the possibility of inserting events with past valid timestamps. Additional rule triggering modes are necessary, since triggering a rule on an event with a past valid time will not always be sensible in an application. A further complication in this situation is the lifecycle. Insertion of an event with a past valid time can cause a violation of the lifecycle by a

sequence of past events. Since history cannot be undone, lifecycles must also specify what happens in case of violations. Clearly, this requires a more sophisticated specification formalism than process algebra, possibly some form of deontic logic [Von Wright, 1972, Meyer and Wieringa, 1991].

The current facilities for evolution in DEGAS only allow object evolution. These facilities can be extended to the class level of DEGAS to implement an aspect of schema evolution, viz., changes of a class' capabilities. Class evolution is more complicated than object evolution due to the number of objects involved, viz., all instances of a class. Enforcement of schema changes is mainly complicated by the existence of multiple class objects. Application of a voting protocol is necessary to avoid schema conflicts between class objects. A further extension of evolution facilities will probably fit better with class membership based on type conformance. The introduction of DEGAS-style schema evolution facilities in a model with conformance-based class membership for autonomous objects forms an interesting proposition. In such an environment, a class informs its members of the update to the class' schema. As a consequence of object autonomy, each member gets the opportunity to respond, whether it will follow the schema update and remain a member of the class, or not.

DEGAS' application designs add capabilities to an object at the time they are needed. This is a consequence of the distinction between inherent and transient capabilities. A further feature of DEGAS is that relation objects are existentially dependent on other objects. Hence, their existence is inherently temporal, meaning that a relation object is also a temporal grouping of capabilities. Furthermore, data stored in relations will often be derived from its partners' data. These features all imply a looser coupling of data and the entities grouping data. This coupling might be further loosened by dropping the distinction between objects and queries. This would lead to a model with primitive data items, as small a single capabilities, and higher level groupings of data. These groupings define a small "schema" of information, that is assembled from primitive data in the system. These data groupings are similar to DEGAS queries in their definition of data derived from primitive data. Leaving the distinction between objects and queries means that queries become autonomous objects too. Hence, the objects representing data groupings assemble the data themselves, like DEGAS relation objects obtain data from the partners in the relation.

Summing up this thesis, we conclude that a database of autonomous objects is a good proposition. It leads to a clean object model, is founded on a straightforward formalisation, and can be implemented on a small base of primitives. Furthermore, it opens up new perspectives for further developments of object-based systems.

Bibliography

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

[Abiteboul and Simon, 1991] Serge Abiteboul and Eric Simon. Fundamental properties of deterministic and non-deterministic extensions of datalog. *Journal of Theoretical Computer Science*, 78:137–158, 1991.

[Abowd, 1996] Gregory Abowd. Software engineering and programming language considerations for ubiquitous computing. *ACM Computing Surveys*, 28(4):190, 1996.

[Agha *et al.*, 1993] Gul Agha, Peter Wegner, and Akinori Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, Cambridge, MA, USA, 1993.

[Agha, 1986] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA, USA, 1986.

[Aiken *et al.*, 1992] Alexander Aiken, Jennifer Widom, and Joseph M. Hellerstein. Behavior of database production rules:termination, confluence, and observable determinism. In *Proceedings of the 1992 ACM SIGMOD International Conference on the Management of Data*, pages 59–68, 1992.

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

[Akker and Siebes, 1995b] Johan van den Akker and Arno Siebes. Applying an advanced data model to graphic constraint handling. In Remco Veltkamp and Edwin Blake, editors, *Proceedings of the 5th Eurographics Workshop on Programming Paradigms in Graphics*, pages 1–16, Maastricht, The Netherlands, September 1995.

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

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

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

[Akker and Siebes, 1997a] J.F.P. van den Akker and A.P.J.M. Siebes. Designing active objects in DEGAS. Technical Report INS-R9702, CWI, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1997. Available through WWW `www.cwi.nl`.

[Akker and Siebes, 1997b] Johan van den Akker and Arno Siebes. DEGAS: A database of autonomous objects. *Information Systems*, 22(2 & 3):121–138, 1997.

[Akker and Siebes, 1997c] Johan van den Akker and Arno Siebes. Enriching active databases with agent technology. In Peter Kandzia and Matthias Klusch, editors, *Proceedings of the First International Workshop on Cooperative Information Agents (CIA'97)*, pages 116–125, Kiel, Germany, 1997. Springer. LNAI 1202.

[Albano *et al.*, 1993] 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.

[Alhajj and Arkun, 1993] Reda Alhajj and M. Erol Arkun. A query model for object-oriented databases. In A.K. Elmargarmid and E.J. Neuhold, editors, *Proc. of the 9th Intl. Conf. on Data Engineering (ICDE'93)*, pages 163–172, Wien, Austria, 1993.

[America, 1987] 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.

[Apers *et al.*, 1992] 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.

[Appelrath *et al.*, 1996] H.-J. Appelrath, H. Behrends, H. Jaspe, and O. Zukunft. Case studies in active database applications. In R. Wagner and H. Thoma, editors, *Proceedings of the 7th International Conference on Database and Expert Systems Applications (DEXA'96)*, pages 69–78. Springer, 1996. LNCS 1134.

[Atkinson *et al.*, 1989] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Deductive and Object-Oriented Databases : Proc. of the 1st Intl. Conf.(DOOD'89)*, pages 223–240, Kyoto, Japan, 1989. North-Holland.

[Baeten and Weijland, 1990] 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.

[Baeten, 1990] J.C.M. Baeten. *Applications of Process Algebra.* Number 17 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1990.

[Bailey *et al.*, 1995] James Bailey, Michael Georgeff, David B. Kemp, David Kinny, and Kotagiri Ramamohanarao. Active databases and agent systems – a comparison. In Timos Sellis, editor, *Proc. of the 2nd Intl. Workshop on Rules in Databases (RIDS'95)*, pages 342–356, Athens, Greece, 1995. Springer. LNCS 985.

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

[Balsters *et al.*, 1993] 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.

[Bancilhon *et al.*, 1992] François Bancilhon, Claude Delobel, and Paris Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Mateo, CA, USA, 1992.

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

[Bell, 1992] Gordon Bell. Ultracomputers: a teraflop before its time. *Communications of the ACM*, 35(8):26-47, 1992.

[Birtwistle *et al.*, 1974] G.M. Birtwistle, O.-J. Dahl, and B. Myhrhaug. *SIMULA begin*. Studentlitteratur, Lund, Sweden, 1974.

[Boncz *et al.*, 1996a] Peter A. Boncz, Fred Kwakkel, and Martin L. Kersten. High performance support for OO traversals in Monet. In R. Morrison and J. Kennedy, editors, *Advances in Databases : 14th British National Conference on Databases (BNCOD14)*, pages 152-169, Edinburgh, UK, 1996. Springer. LNCS 1094.

[Boncz *et al.*, 1996b] Peter A. Boncz, Wilko Quak, and Martin L. Kersten. Monet and its geographic extensions: a novel approach to high performance GIS processing. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in database technology - EDBT '96*, pages 147-166, Avignon, France, 1996. Springer. LNCS 1057.

[Bos and Laffra, 1991] Jan van den Bos and Chris Laffra. PROCOL: A concurrent object-language with protocols, delegation and persistence. *Acta Informatica*, 28:511-538, September 1991.

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

[Cardelli, 1984] 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.

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

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

[Catell, 1994] R.G.G. Catell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, CA, USA, 1994.

[Ceri and Widom, 1990] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. In D. MacLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 566-577, 1990.

[Ceri *et al.*, 1996] Stefano Ceri, Pietro Fraternali, Stefano Paraboschi, and Letizia Branca. *Active Rule Management in Chimera*, chapter 6 in [Widom and Ceri, 1995]. 1996.

[Chen, 1976] Peter Chen. The entity-relationship model: Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9-36, 1976.

[Choenni *et al.*, 1996] R. Choenni, M.L. Kersten, J.F.P. van den Akker, and A. Saad. On multi-query optimization. Technical Report CS-R9638, CWI, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1996. Available through WWW from `www.cwi.nl`.

[Claramunt and Thériault, 1995] Christophe Claramunt and Marius Thériault. Managing time in gis: An event-oriented approach. In James Clifford and Alexander Tuzhilin, editors, *Recent Advances in Temporal Databases - Proc. of the Intl. Workshop on Temporal Databases*, Workshops in Computing, pages 23–42. Springer, 1995.

[Codd, 1970] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

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

[Dayal *et al.*, 1988a] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, and S. Sarin. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

[Dayal *et al.*, 1988b] 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.

[Deux, 1990] O. Deux. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

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

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

[Garcia-Molina and Kogan, 1988] 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.

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

[Gatziu and Dittrich, 1993] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system. In Norman W. Paton and M. Howard Williams, editors, *Rules in Database Systems, Proc. of the 1st Intl. Workshop (RIDS'93)*, Workshops in Computing, pages 23–39, Edinburg, Scotland, UK, 1993. Springer.

[Gatziu and Dittrich, 1994] Stella Gatziu and Klaus R. Dittrich. Detecting composite events in active database systems using Petri nets. In *Proc. of the 4th Intl. Workshop on Research Issues in Data Engineering (RIDE-ADS'94)*, pages 2–9, Houston, USA, 1994.

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

[Gehani *et al.*, 1992] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In M. Stonebraker, editor, *Proc. of the 1992 ACM SIGMOD International Conference on the Management of Data*, pages 81–90, San Diego, USA, 1992.

[Geurts *et al.*, 1990] Leo Geurts, Lambert Meertens, and Steven Pemberton. *ABC Programmers Handbook.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, 1990.

[Ginsburg, 1993] Seymour Ginsburg. *Object and Spreadsheet Histories*, chapter 12 in [Tansel *et al.*, 1993]. 1993.

[Glasson *et al.*, 1994] Bernard C. Glasson, Igor T. Hawryszkiewycz, and B. Alan Underwood, editors. *Business process re-engineering: information systems opportunties and challenges - Proc. of the IFIP TC8 open conference*, IFIP transactions A: computer science and technology, Gold Coast, Queensland, Australia, 1994. North-Holland.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, Reading, MA, USA, 1983.

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

[Hammer and Champy, 1993] Michael Hammer and James Champy. *Reengineering the Corporation: A Manifesto for Business Revolution.* Harper-Collins, 1993.

[Hammer, 1990] Michaell Hammer. Re-engineering work: Don't automate, obliterate! *Harvard Business Review*, July 1990.

[Hanson, 1989] 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.

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

[Hofmann-Wellenhof *et al.*, 1994] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins. *Global Positioning System : Theory and Practice.* Springer, third edition, 1994.

[IEEE, 1994] IEEE Computer Society. *Proc. of the 1st Intl. Conf. on Requirements Engineering*, Colorado Springs, CO, USA, 1994.

[Imielinski and Badrinath, 1992] T. Imielinski and B.R. Badrinath. Querying in highly mobil distributed environments. In *Proc. of the 18th Conf. on Very Large Data Bases (VLDB'92)*, pages 41–52, Barcelona, Spain, 1992.

[Imielinski and Badrinath, 1994] Tomasz Imielinski and B.R. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM*, 37(10):19–28, October 1994.

[ISO, 1994] International Standards Organisation. *ISO-ANSI Working Draft: Database Language SQL3*, 1994. X3H2/94/080 and SOU/003.

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

[Kappel and Schrefl, 1996] Gerti Kappel and Michael Schrefl. Modeling object behavior: To use methods or rules or both? In R. Wagner and H. Thoma, editors, *Proceedings of the 7th International Conference on Database and Expert Systems Applications (DEXA'96)*, pages 584–602. Springer, 1996. LNCS 1134.

[Kersten, 1991] Martin L. Kersten. Goblin: a DBPL designed for advanced database applications. In Dimitris Karagiannis, editor, *Database and Expert Systems Applications, Proceedings of the International Conference*, pages 354–349, Berlin, Germany, 1991. Springer.

[Kim, 1995] Won Kim. *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press/Addison-Wesley, New York, USA, 1995.

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

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

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

[Loucopoulos, 1994] P. Loucopoulos, editor. *Entity-Relationship Approach - ER'94 : business modeling and re-engineering*, Manchester, UK, 1994. Springer. LNCS 881.

[Lutz, 1996] Mark Lutz. *Programming Python.* O'Reilly & Associates, 1996.

[Maier and Stein, 1987] David Maier and Jacob Stein. Development and implementation of an object-oriented dbms. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

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

[McKenzie and Snodgrass, 1991] 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.

[Meyer and Wieringa, 1991] John-Jules Meyer and Roel J. Wieringa, editors. *Deontic logic in computer science : normative system specification (selected papers from the first international workshop on deontic logic in computer science (DEON'91))*, Chichester, UK, 1991. Wiley.

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

[Morgenstern, 1983] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proc. of the 9th Intl. Conf. on Very Large Data Bases (VLDB'83)*, pages 34–42, Firenze, Italy, 1983.

[Navathe and Ahmed, 1993] Shamkant B. Navathe and Rafi Ahmed. *Temporal Extensions to the Relational Model and SQL*, chapter 4 in [Tansel *et al.*, 1993]. 1993.

[Nijssen and Halpin, 1990] 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.

[OMG, 1996] The Common Object Request Broker: Architecture and Specification - version 2.0. Technical Report PTC/96-08-04, Object Management Group, July 1996.

[Özsu and Straube, 1991] Tamer Özsu and Dave D. Straube. Issues in query model design in object-oriented database systems. *Computer Standards & Interfaces*, 13:157–167, 1991.

[Porter, 1985] Michael E. Porter. *Competitive Advantage: Creating and Sustaining Superior Performance.* Free Press, 1985.

[Reisig, 1985] Wolfgang Reisig. *Petri Nets: An Introduction.* Monographs in Theoretical Computer Science. Springer, Berlin, Germany, 1985.

[Richardson and Schwarz, 1991] 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.

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

[Rossum, 1995a] Guido van Rossum. Python library reference. Technical Report CS-R9524, CWI, Amsterdam. The Netherlands, 1995. Available through the World Wide Web from `www.cwi.nl`.

[Rossum, 1995b] Guido van Rossum. Python reference manual. Technical Report CS-R9525, CWI, Amsterdam. The Netherlands, 1995. Available through the World Wide Web from `www.cwi.nl`.

[Rossum, 1995c] Guido van Rossum. Python tutorial. Technical Report CS-R9526, CWI, Amsterdam. The Netherlands, 1995. Available through the World Wide Web from `www.cwi.nl`.

[Rosenschein and Zlotkin, 1994] Jeffrey S. Rosenschein and Gilad Zlotkin. Designing conventions for automated negotiation. *AI Magazine*, 15(3):29–46, 1994.

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

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

[Sciore, 1991] 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.

[Shaw and Zdonik, 1990] Gail M. Shaw and Stanley B. Zdonik. A query algebra for object-oriented databases. In M.T. Liu, editor, *Proc. of the 6th Intl. Conf. on Data Engineering (ICDE'90)*, Los Angeles, CA, USA, 1990.

[Sheth and Larson, 1990] 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.

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

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

[Simon and de Maindreville, 1988] Eric Simon and Christophe de Maindreville. Deciding whether a production rule is relational computable. In *Proceedings of the ICDT 88*, LNCS 326, pages 205–222. Springer, 1988.

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

[Snodgrass, 1994] Richard T. Snodgrass, (chair). TSQL2 language specification, 1994. Available by FTP from ftp.cs.arizona.edu.

[Stonebraker *et al.*, 1996] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[Stonebraker, 1986a] Michael Stonebraker. *The INGRES papers: anatomy of a relational databases system.* Addison-Wesley, 1986.

[Stonebraker, 1986b] 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.

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

[Tanenbaum, 1996] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, NJ, USA, third edition, 1996.

[Tansel *et al.*, 1993] Abdullah Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, Redwood City, CA, USA, 1993.

[Ungar and Smith, 1987] David Ungar and Randall B. Smith. Self: The power of simplicity. In Norman K. Meyrowitz, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *SIGPLAN Notices*, pages 227–242, Orlando, FL, USA, December 1987.

[Von Wright, 1972] Georg Henrik Von Wright. *An essay in deontic logic and the general theory of action.* North-Holland, Amsterdam, The Netherlands, 1972.

[Voort, 1994] M.H. van der Voort. *A Design Theory for Database Triggers.* PhD thesis, Universiteit van Amsterdam, The Netherlands, September 1994.

[Vreeze, 1991] 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.

[Weiser, 1993] Mark Weiser. Some conputer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, July 1993.

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

[Widom and Finkelstein, 1990] 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.

[Widom *et al.*, 1991] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 275–285, 1991.

[Widom, 1996] Jennifer Widom. *The Starburst Rule System*, chapter 4 in [Widom and Ceri, 1995]. 1996.

[Wieringa *et al.*, 1995] 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.

[Wieringa, 1996] R.J. Wieringa. *Requirements Engineering: Frameworks for Understanding*. John Wiley & Sons, 1996.

[Wooldridge and Jenning, 1995] Michael Wooldridge and Nicholas R. Jenning. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[Workflow Management Coalition, 1996] Workflow Management Coalition. Terminology and Glossary WFMC-TC-1011 2.0. Available through WWW at `www.wfmc.org`, June 1996.

[Wuu and Dayal, 1992] 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.

Index

`Broadcast`, 110
`CheckSelector`, 97, 129
`DistributeQuery`, 119
`ExecuteMethod`, 99, 128
`Extend`, 102, 129
`Kill`, 103, 129
`MethodAllowed`, 99, 129
`NewEmptyObject`, 109
`New`, 111
`Remove`, 103, 129, 130
`getPathExpr`, 100
`get`, 100, 129
`initiate`, 117
`queryResult`, 120
`sendMessage`, 109
`sendQuery`, 109
`sendReply`, 110
`shipResult`, 120

active databases, 23
active DBMS prototypes, 24
addon, 44
addon class object, 112
addon dropping, 83
addon extension, 83, 129
addon set, 83
assignment, 72
attribute set, 95, 127
attribute value, 77
attributes, 44, 55
autonomy, node, 19
autonomy, object, 19

basic DEGAS object, 94, 125
basic functions, 53
basic types, 53, 64
business process reengineering, 18
business rules, 25

chain information system, 16
class header, 54
class object, 45, 111
clock, 74
compensating tasks, 185
complex objects, 31
compound statement, 73
condition, 82
confluence, 196
coupling modes, 27

DEGAS acronym, 9
DEGAS database, 89
DEGAS implementation, 123
DEGAS object, 43
DEGAS query, 58, 89
DEGAS syntax, 53
DEGAS system layer, 107, 134
DEGAS^{-} model, 188
DEGAS^{-2} model, 192
data dictionary, 146
data entry, 136
decidability, 199
design theory, 187
distribution model, 60
domains, 65

ECA format, 25
encapsulation, 31
encapsulation of control, 20
encapsulation of execution, 19
encapsulation of specification, 19
encapsulation, maximal, 19
event detection, 131
event expression, 79
event specification, 25
event triggering, 81
execution cycle, 104

federated databases, 20
fragmentation data, 147

generalisation, 44
guard condition, 48

hermit type, 64
historical databases, 29
history, event, 76
history, state, 75, 94, 126
history, valid object, 89

impedance mismatch, 33
independence, 197
inherent capabilities, 44
instance-oriented semantics, 27
intelligent agents, 217
interpretation, 76

lifecycle, 44, 57, 84, 127
lifecycle checking, 87
lifecycle composition, 85
lifecycle semantics, 84
lifecycle set, 97, 127

massively parallel computing, 16
matching sub-history, 79
meta class, 45
method queue, external, 95, 127
method queue, internal, 95, 127
method semantics, 71
method set, 95, 127
method typing, 67
methods, 44, 55
mobile computing, 16
model for a database, 77
modularisation, 34, 157

nested query, 59, 90
network, 60

object communication, 134
object creation, 134
object identity, 31
object-oriented databases, 30

problem statement, 9
process algebra, 36
Python, 124

query interface, 137
query processing, 142
query quality, 59, 148
query queue, 95, 127
query result, 90

reachability, 90
relation class object, 117
relation creation, 139
relation object, 115
relation objects, 45
reply box, 95, 127
role, 39
rule execution, 26, 106
rule set, 97, 127
rule triggering, 88
rulebase, 34
rules, 44, 56, 88

schema information, 147
selector, 58, 79
set iteration, 72
set-oriented semantics, 26
site, 60
site object, 61, 118
snapshot interpretation, 71
snapshot state, 75
SQL3, 35
strong agency, 218
subtyping, 65

temporal databases, 28
temporal dimensions, 28
termination, 196
termination in n steps, 197
time window, 59, 80
timestamp, 74
transient capabilities, 44
tuple type composition, 68
type context, 69
type system, 63
typical database state, 199

ubiquitous computing, 217, 223
underlying type, 64

variant interpretation, 71

weak agency, 218
willingness, 90
workflow, 157, 162
workflow evolution, 183
workflow management systems, 157
workflow routing, 162, 164
workflow schema, 162

Nederlandse Samenvatting

Het onderzoek gerapporteerd in dit proefschrift vond plaats op het gebied van databases, ofwel gegevensbanken. De centrale vraagstelling ging over een database met maximale autonomie voor de componenten:

1. Is een dergelijk database model eenvoudig te realiseren?
2. Is het hanteerbaar voor de ontwerper van een applicatie?
3. Is het op heldere wijze te formaliseren?

Deze vraagstelling wordt in alle aspecten positief beantwoord door de constructie van DEGAS, een database taal gebaseerd op autonome objecten.

De behoefte aan systemen van autonome componenten ontstaat door een aantal ontwikkelingen. In de informatietechnologie zien wij een ontwikkeling naar systemen gebaseerd op netwerken van mobiele computers. Bovendien ontstaan er grote systemen, die draaien op computers met een groot aantal processoren. In dit soort omgevingen zal het zeer moeilijk zijn centrale coördinatie te voeren over alle componenten. Daarnaast verandert de manier waarop organisaties informatiesystemen gebruiken. In toenemende mate worden informatiesystemen van verschillende organisaties geïntegreerd. Als gevolg daarvan ontstaan systemen waarvan de onderdelen verschillende eigenaars hebben.

Deze ontwikkelingen leiden er toe dat centrale coördinatie hetzij niet mogelijk is, hetzij niet wenselijk is. Daarom is er een behoefte aan systemen gebaseerd op *autonome objecten.* Dit zijn gegevensobjecten, waar alle kennis over een object in het object zelf gedefinieerd is. Bovendien is een autonoom object niet onderworpen aan enige vorm van centrale controle.

Deze aanname vormt de basis van DEGAS. Een DEGAS database bestaat uit objecten. Ieder object bevat de definitie van een element uit het toepassingsdomein van de database, bijvoorbeeld een bankrekening. Een object heeft bepaalde attributen, die de gegevens over het object bevatten. Voorbeelden voor een bankrekening object zijn het saldo en de maximum limiet voor een negatief saldo. Daarnaast wordt het gedrag van het object gedefinieerd. Dit gedrag bestaat uit de mogelijke acties van het object, methoden genoemd. Voor een bankrekening zijn dit acties als het crediteren en debiteren van de rekening.

DEGAS voegt nog een tweetal dimensies van gedrag toe aan traditionele objectgeoriënteerde databases. De "lifecycle" definieert een ordening en condities op de uitvoering van acties door het object. Een voorbeeld is de eis dat er nooit meer dan één keer per maand geld mag worden opgenomen van een spaarrekening.

Daarnaast specificeren regels dat bepaalde acties moeten worden uitgevoerd in reactie op een bepaalde situatie. Deze is gedefinieerd door een "event", de

uitvoering van bepaalde acties door het object, en door een conditie op de toestand van de database. Een voorbeeld is een automatische overschrijving van een betaalrekening naar een spaarrekening, als het saldo van de bankrekening groter dan Hfl.10.000 is. Gezamenlijk noemen wij attributen, methoden, lifecycle en regels ook wel de capaciteiten van het object.

Verbanden tussen objecten worden gemodelleerd door relaties. Een relatie is zelf ook een object, om gegevens en gedrag van de relatie te kunnen opslaan in de database. Een voorbeeld van een relatie tussen een persoon en een bank is de bankrekening.

De capaciteiten van een DEGAS object kunnen worden uitgebreid met behulp van "addons". Een addon definieert een aantal capaciteiten, die toegevoegd kunnen worden aan een object, tijdens zijn bestaan. Het voornaamste gebruik van addons is het modelleren van de rollen van objecten in relaties. Een relatie geeft een object nieuwe capaciteiten. Zo geeft een bankrekening een persoon de mogelijkheid giraal te betalen. Een persoon object zal daarom worden uitgebreid met een rekeninghouder addon, als het een bankrekening opent.

De bijdragen van DEGAS aan de ontwikkeling van database management systemen kunnen als volgt samengevat worden:

1. **Modularisatie van actieve databases.** Door de consequente toepassing van het encapsulatie principe zijn regels in de database op dezelfde manier gemodulariseerd als de data. Dit zorgt voor een beter inzicht in de gedefinieerde regels.

2. **Object Evolutie.** In veel toepassing zal een object zich tijdens zijn levensduur ontwikkelen. Dit betekent dat er dynamisch capaciteiten toegevoegd en weggelaten moeten kunnen worden. DEGAS biedt hiervoor een faciliteit in de vorm van het addon-mechanisme.

3. **Formalisatie en integratie van actieve en historische databases.** Het gebruik van proces algebra voor zowel de specificatie van event expressies, als de specificatie van de historie, leidt tot één formalisatie van actieve en historische databases. Met name de directe definitie van de semantiek van regels in termen van de historie is een voordeel.

4. **Queries in een actieve database.** De functionaliteit in actieve databases om event patronen te herkennen in de geschiedenis van een object kan ook gebruikt worden om vragen te stellen aan de database. Dit maakt het mogelijk om te refereren aan historische situaties zonder het tijdstip daarvan precies te hoeven weten.

5. **Onbeslisbaarheid resultaten voor regels.** In dit proefschrift hebben wij bewezen, dat terminatie en confluentie van regelverzamelingen onbeslisbare predicaten zijn voor een subtaal van DEGAS. Dit betekent dat deze predicaten ook onbeslisbaar zijn voor DEGAS zelf.

De voordelen van DEGAS bij het modelleren van toepassingen laten wij zien door middel van een voorbeeld. Het gekozen voorbeeld is "workflow management", het sturen van gecomputeriseerde gegevensstromen binnen een organisatie. DEGAS voldoet om alle internationaal gestandardiseerde elementen van workflow management te modelleren. Bovendien zorgen de modelleerconcepten van DEGAS voor een flexibel model, waar de benodigde sturings- en toepassingsinformatie op het juiste moment beschikbaar is.

Een ander aspect van het ontwerpen van actieve databases is het verifiëren van gewenste eigenschappen van verzamelingen regels. Een voorbeeld van zo'n eigenschap is terminatie, dat wil zeggen, de eigenschap dat elke executie van een verzameling regels eindig is. In dit proefschrift laten wij zien dat het reeds voor zeer eenvoudige regels niet meer mogelijk is om deze eigenschap te voorspellen.

Tenslotte geven wij aan het einde van dit proefschrift nog een vooruitblik op de toekomst van actieve databases. Deze zal liggen in een verwevenheid met zogenaamde intelligente agenten, zelfstandig functionerende software voorzien van kunstmatige intelligentie. Het toepassen van deze technologie in databases biedt interessante voordelen. Verder laten wij zien dat beperkte uitbreidingen voldoen om DEGAS te laten functioneren als programmeertaal voor intelligente agenten.

Curriculum Vitae

Johan van den Akker was born on October 9th, 1969 in Gouda. He did his final exam Gymnasium β at Rythovius College in Eersel in 1988. Johan studied Business Oriented Computer Science at Erasmus University in Rotterdam from 1988 to 1993. He wrote his Master's thesis on a paraconsistent default logic, advised by Dr Yao Hua Tan. Results from this thesis were published at the Dutch Artificial Intelligence Conference, the Brazilian Artificial Intelligence Symposium and in the journal *Logique et Analyse.*

After his graduation, Johan went to the Department of Algorithmics and Architecture (later transformed to the Department of Information Systems) of the Centrum voor Wiskunde en Informatica, the Dutch National Centre for Mathematics and Computer Science. As an *Onderzoeker in Opleiding*, he did research into active object databases, that led to this thesis. He currently works as a Research Consultant for ID Research, a consulting firm in Gouda, the Netherlands.