FORMAL SPECIFICATION AND MODELING OF COMPLEX SYSTEMS

TOWARDS A PHYSICS OF INFORMATION VIA NETWORKS

Dissertation

zur Erlangung des Doktorgrades

des Fachbereichs Physik

der Universität Hamburg

vorgelegt von

THOMAS TOMKOS

aus Reinbek

Hamburg

1999

Gutachter der Dissertation:	Prof.Dr. G. Mack Prof.Dr. K. Fredenhagen
Gutachter der Disputation:	Prof.Dr. G. Mack Prof.Dr. J. Bartels
Datum der Disputation:	30. April 1999
Sprecher des Fachbereiches Physik und Vorsitzender des Promotionsausschusses:	Prof.Dr. B. Kramer

"...nicht einfach die Menge der strukturell ermöglichten Relationen, sondern deren Selektivität: auch nicht nur ein (empirisch gesicherter) Erkenntniszusammenhang zwischen den Variablen Grösse und Strukturiertheit, sondern die Relation zwischen positiver Bestimmung der Grösse und negativer Bestimmung des Ausscheidungseffekts der Struktur. Die Komplexität hat ihre Einheit also in der Form einer Relation: in der Relation wechselseitiger Ermöglichung von Elementmengen und reduktiven Ordnungen."

LUHMANN, 1975 [1]

Abstract

Complex systems are investigated in numerous disciplines. A central role is played by the used modeling methods, since they influence the structure and dynamics of models. Based on an interdisciplinary systems theory, in the thesis it is investigated to what extend the structure and dynamics of system may be modeled by networks with universal, local operations defined on them. In particular, physical systems, like classical electrodynamics, and information systems are investigated. It is analyzed, how conventional models may be fitted into the framework. An analysis of the metamodel of the most important recent object-oriented modeling approach UML, shows how a network structure for information systems may be manifested. The multiscaling properties of the UML are identified. A comparison of all methods dealt with based on a recent classification scheme is given.

Zusammenfassung

Komplexe Systeme sind Gegenstand zahlreicher Disziplinen. Eine zentrale Rolle spielen dabei die verwendeten Modellierungsmethoden, da diese entscheidend Einfluss auf Struktur und Dynamik von Modellen nehmen. In der Arbeit wird auf Basis einer interdisziplinären Systemtheorie untersucht, inwieweit sich Struktur und Dynamik komplexer Systeme als Netzwerke mit universellen, lokalen Operationen auf ihnen modellieren lassen. Inbesondere werden dazu physikalische Systeme, wie klassische Elektrodynamik, und Informationssysteme untersucht. Es wird analysiert, wie sich konventionelle Modelle in den gewählten Rahmen eingliedern lassen. Eine Analyse des Metamodells des gegenwärtig favorisierten objekt-orientierten Modellierungsansatzes UML zeigt auf, wie sich eine Netzwerkstruktur für Informationssysteme manifestiert. Die Mehrskalen-Eigenschaften der UML werden identifiziert. Ein Vergleich aller behandelten Methoden auf Basis eines aktuellen Klassifizierungsschemas findet statt.

Contents

1 Introduction and Motivation 3
- 1.1 Purpose and Structure of the Thesis . . . 4
- 1.2 General Remarks . . . 6

2 Theory of Complex Systems (ToCS) 13
- 2.1 Categories and Functors . . . 14
- 2.2 Representations . . . 18
- 2.3 Systems . . . 20
- 2.4 Gauge Transformations . . . 25
- 2.5 Dynamics of Systems . . . 28
- 2.6 Composite Objects . . . 32
- 2.7 Summary . . . 39

3 ToCS-based Formal Modeling 41
- 3.1 Formal Models . . . 41
- 3.2 Modeling Behavior . . . 44
  - 3.2.1 State-Transition Systems . . . 44
  - 3.2.2 Petri Nets . . . 45
- 3.3 Physical Laws and Systems Theory . . . 58
  - 3.3.1 Classical Electrodynamics . . . 59
  - 3.3.2 The Dirac equation . . . 67
- 3.4 Summary . . . 69

4 Complex Information Systems 71
- 4.1 Complexity in Information Systems . . . 72
- 4.2 Object-Oriented Analysis . . . 76
- 4.3 Object-Oriented Development Processes . . . 85
- 4.4 Object-Orientation and ToCS . . . 88
- 4.5 Summary . . . 95

5 Analysis of the UML Metamodel 97
- 5.1 Metamodeling . . . 98
- 5.2 Language Specification . . . 103

5.3 Large Scale Structure of the Metamodel . . . 106
5.3.1 Model and Model Representation . . . 106
5.3.2 Horizontal Structuring Method . . . 108
5.3.3 Common Mechanisms in the Metamodel . . . 113
5.4 Modeling Structural Properties . . . 114
5.4.1 The Core . . . 116
5.4.2 The Auxiliary Elements package . . . 124
5.4.3 Modeling Extensibility . . . 125
5.5 Modeling Dynamics . . . 127
5.5.1 The Common Behavior Package . . . 127
5.5.2 Societies of Objects: Collaborations . . . 131
5.5.3 Composition of Collaborations . . . 134
5.5.4 Statecharts and Object State Machines . . . 135
5.5.5 Conflicting Transitions . . . 143
5.5.6 Activity Models . . . 144
5.5.7 View from the Outside: Use Cases . . . 145
5.6 Summary . . . 148

6 Discussion 151
6.1 Identification of Multiscaling Properties . . . 151
6.2 Classification and Comparison . . . 153
6.3 Summary . . . 159

7 Conclusions and Outlook 161

A ToCS/SyCL 163
A.1 Beyond Analysis . . . 163
A.2 Implementing ToCS: SyCL . . . 164
A.3 System Development with ToCS/SyCL . . . 167

B Constructing the Dirac System 169
B.1 The Dirac System . . . 170
B.2 The Dirac Equation . . . 177

C The UML Notation 181
C.1 Views and Diagrams . . . 181
C.2 Graphical Symbols . . . 182

D Two Proofs 183
. . .
Bibliography . . . 187

Chapter 1

INTRODUCTION AND MOTIVATION

Suppose, some day a system-engineer is coming to you, asking the following:

> *" Well, being a theoretical physicist, maybe you're able to help us.*
> *I know that you investigate models of complex systems, like spin-glasses, and you use tools like multiscale analysis to handle their complexity. So, I know that you need to understand the dependency of your systems on initial conditions, criticality, the arising of composite structures, like clusters, finally, you need to understand all kinds of emergent phenomena.*
>
> *All right, I believe that you may find our problem interesting as well. We are constructing a software intense information system, and there are a large number of people simultaneously involved in it. To avoid the uncontrolled diffuse of the system, there are standardized interfaces for its extension, which enable the modification of the system by a few predefined mechanisms.*
> *Nevertheless, I do not understand the behavior of the system. Some changes remain local in their effects, while others run through the system like cascades, changing data almost everywhere, while in a worst case cause the break-down of the whole system.*
> *However, I am sure that is not due to the specific algorithms we use in small software pieces. Rather, I believe that this effect is caused by the interrelations of the software components, that is, by the network architecture. You know, parts of the software do hide their internal structure, while interacting with other parts only via specific channels. They receive information, process them internally, and transmit themselves information to other components. Based upon such 'software-modules', it is natural to expect phenomena that remind one to the ones you are focused on. Therefore, it would be extremely useful for us to know, if for example, by building new composite modules, it is possible, to consolidate some interactions, while hiding others as internal mechanisms,*

> *turning thereby the system into a more stable one. But, this requires theoretical simulations of the underlying network architecture, to understand the system's behavior in terms of its modules."*

Well, you think that this is a somehow crazy, but interesting task. At least, wouldn't this enlarge the applicability of your own methods? So you agree...
... and get horrified the other day by receiving his 'system' : a package with a heap of paper and CD-ROMs. Although very eager to work out his problem, it seems impossible for you to make a somehow quantitative analysis of 18 millions lines of code! This fails in any way from being an ordinary network structure, with precise and well-defined objects and relations!

1.1 PURPOSE AND STRUCTURE OF THE THESIS

The scenario described above may be summarized in the following fundamental question, which serves a motivation for the thesis:

> *Is it possible to analyze information systems by using well-known concepts of discrete and statistical physics?*

Usually, the answer will be "*no*". In physics, the used modeling techniques apply to systems, whose participating elements and their relations are given by well-defined entities, which allow the formulation in precise mathematical terms. On the other hand, information systems usually exist in form of programming code, or at worst, in a loose description, which does not allow to apply immediately formal techniques for their analysis.

This work is an attempt, to turn the "*no*" into a somehow more fruitful "*yes, maybe*". The basic idea here is, to establish an interconnection between these areas by considering both physical systems, as well as information systems as networks, which are described by using the same modeling framework. For this, the *Theory of Complex Systems* (ToCS) initiated by G. Mack, is used as a general framework for an abstract formulation of the concepts that occur in almost every network structure. Based on this framework, it is shown how models of physical systems may be formulated and analyzed. It is argued how elementary modeling techniques for information systems may be directly formulated within our approach. The incorporation of more advanced information systems into our framework requires the usage of the object-orientation paradigm to establish a underlying network structure.

Due to the immense growth of complexity in information systems, the need for precise system modeling techniques methods becomes more and more important. Since a major aspect in this context is to assure the safety and consistency of a system, formal modeling methods are considered right now to be the only way to handle this tasks properly. Beside a

lot of naive and heuristic modeling techniques that have been used not only in computer sciences, but also in other areas of science and engineering in the past, many formal or at least semi-formal modeling methods to handle system complexity precisely have been developed in the last years. The most powerful modeling approach that incorporates object-orientation is the recent Unified Modeling Language, which is analyzed here to provide a substantial foundation for the required network structure.

Structure of the thesis

The following section 1.2 presents some general remarks concerning the occurrence and definition of systems and their properties. The interdisciplinary usage of systemic investigations is sketched, and the role of the presented Theory of Complex Systems is outlined in this context.

In chapter 2 the used analytical Theory of Complex Systems is presented. The basic ingredients, like categories, functors, and their representations are explained. It is shown that gauge transformations arise naturally in this general setting by natural transformations to the identity functor. Locality is introduced as a concept that turns the category structure into a system structure. It is described what kinds of dynamics may be defined generically in any system. Finally, a precise definition for composite objects is suggested. The definition is corroborated by some examples for the occurrence of composite objects.

Based upon the concepts of the first chapter in chapter 3 the ToCS-based modeling approach of both physical systems, and information systems is illustrated. For this, standard modeling techniques for information systems, like state-transition-systems and P/T-nets are introduced and related to ToCS. In particular, it is shown, how the structure and dynamics of P/T-nets may be achieved in a ToCS-based formulation. Some advantages of this treatment are outlined. On the other side, the framework is used to describe classical electrodynamics as generic system theoretical dynamics, and the ToCS-based formulation of the massless Dirac-equation is given. The specific details of the construction of the system and dynamics of the latter are presented in appendix B.

In chapter 4 information systems are considered as examples for systems with generic complexity. After explaining the role of complexity in information systems, it is argued that object-oriented analysis is a powerful method to handle their complexity. This is achieved by using a underlying network structure to formulate the system dynamics as a dynamical network of communicating objects. Some development methodologies are sketched. The chapter provides a foundation of object-oriented analysis that is easy to incorporate in a ToCS approach and builds thereby an essential connection. This is explicitly shown in the last section of this chapter.

In chapter 5 the UML-modeling approach is investigated as a powerful specification of advanced object-oriented analysis. The chapter explains the object-oriented metamodeling

approach. It describes the loose metamodel architecture of the UML and illustrates, how its specification is realized in the original semantics document of the UML. The metamodel core of the UML is identified. A thorough analysis of all model elements that may be used in the UML follows. It is explained, how the metamodel may be extended to incorporate model specific properties, and it is shown, how the dynamics of a system is modeled in the UML. Some corrections to the original specification are made.

The discussion in chapter 6 identifies the main multiscaling concepts of the UML. They are used in the UML for complexity reduction. Further, all modeling techniques presented in the thesis are compared using a recently proposed classification and integration paradigm for system specification techniques. This comparison embeds the ToCS-based approach into the field of formal system specification.

Chapter 7 summarizes the results of the thesis and provides a brief outlook.

In appendix A, the implementation of ToCS as an object-oriented class library (SyCL) is outlined. Appendix B contains the detail of the Dirac-system construction, while appendix C states some remarks on the UML-notation. Appendix D contains proofs of the representation theorems of chapter 1.

1.2 GENERAL REMARKS

Nowadays almost every scientific or semi-scientific discipline[1] refers to all or at least to parts of its own subject as being a complex system. A great amount of work and literature in various disciplines illustrates this trend impressively [2]. Since concepts in science have always been strongly related to ideas and beliefs present in the common consciousness of their time, this trend is not surprising in the age of networks and in particular of the Internet and complex information systems, like the vast-size ERP-Systems[2].

Systems and multi-scale phenomena

However, a system theoretical approach may be of major relevance in various contexts. It enables first the representation of structural issues on different scales of consideration and second the handling of a large number of mutual dependencies of the participating elements. In this sense, it concentrates on the development of descriptions of natural processes as models of interconnected constituents or in other words, as dynamical networks of objects. The interrelations between objects are abstractly considered to represent communication channels enabling thereby the transmission and receiving of information between objects. Different systems are distinguished due to the structural properties of the relations in them.

[1] or **subsystem** in the sense of Niklas Luhmann († 06.XI.1998)

[2] ERP: Enterprise Resource Planning, for example the $18 \cdot 10^6$ lines of code SAP R/3-system [3].

There are two major concepts that must be distinguished, when formulating a systemic framework. First a definition for general systems accepted in different disciplines must be given. This can be realized more or less formally, depending on its future usage in the subject under consideration. Second, a notion of complexity must be introduced, defining the properties that characterize complex systems in contrast to simple systems. Specifying both aspects enables one to proceed into the analysis of structural properties and of the dynamical behavior of the system, the later being a highly sophisticated task.
The definition of a system is widely accepted, although there is some freedom as to what extend the definition should be formalized. A useful consensus that is actually accepted in commercial scenarios is the **DIN 19226** definition, which may be used also as a minimal-formalism definition in other contexts:

> "A *system* is a construction, which is delimited or thought as delimited from its environment by a hull and which consists of a set of *elements* (constituents or subsystems) that influence each other by fixed *relations* between them." [3]

For the purposes considered in this thesis, an important modification must be remarked. The systems considered here are adaptive systems, that is, they possess inherent dynamics that do not allow to treat the relations as fixed in general. However, as stated in parenthesis, it is important that the elements of a system may be systems themselves. This property enables a very natural treatment of phenomena arising on different levels of abstraction. Objects on a certain scale that are builded or at least correspond to a collection of objects on a finer scale are called composite objects. Although it seems to be a feasible task, these aspects involve some of the most challenging and fascinating problems complex systems possess. Since composite objects may be interconnected to its constituents and also among each other by relations, the question arises if and how these relations are related to relations on finer scales. Further, since complex systems perform dynamical evolutions it is a highly non-trivial issue to investigate how composite objects behave under the dynamics. Moreover, it is an central question, if composite objects may arise dynamically, that is, by their coherent behavior.

Complex systems in physics

In physics and mathematics, complex systems usually arise when the number of participating elements becomes very large and does not allow an independent analysis of the system based on small subsystems. This is the case, if most of the objects interact with each other, and almost none of the interactions may be neglected or approximated by simple superposition of others. Such systems are described by a huge number of coupled differential equations. Systems with such properties give rise to macroscopic phenomena (emergent behavior) originating in the microscopic interactions of its elements as for example in ordinary statistical mechanics. Further complexity specific properties like chaotical behavior

[3] Original german text: *Ein System ist ein Gebilde, das durch seine Hülle von seiner Umgebung abgegrenzt oder abgegrenzt gedacht ist und aus einer Menge von Elementen (Bestandteilen oder Teilsystemen) besteht, die über feste Beziehungen zwischen ihnen aufeinander einwirken.*

and self-organization do arise in this context, too.

Actually, complex systems are investigated in physics mainly in the context of nonlinear systems, leading to complexity in form of chaos, or synergetic effects [4]. Different scales and generic emergent phenomena are usually investigated in simple models, like magnetism in the Ising-model or self-organized criticality in the BTW-model [5]. In particular, the effect of self-organized criticality is an essential idea to understand effects arising in almost every complex system [6] and are investigated also experimentally [7].

It is important to note that since complex systems are investigated in physics from the point of view of dynamical systems they are treated usually with conventional analytical or numerical methods, like large scale computing or multi-scale analysis. Unfortunately this is only possible, if the system allows a formalization up to this point. Since this is not always intended and often not a feasible task, it is important to understand at least how systems must qualitatively be specified, that is, what information plays an essential role in understanding their structure and how they can be modeled from a dynamical point of view [8].

Another main field of use for systems are all neural networks applications, since naively spoken systems may be considered as their generalizations allowing much more interactions.

Interdisciplinarity

Systems play an essential role in various contexts and disciplines. Since their role varies from discipline to discipline, this leads to fruitful interactions that underline the interdisciplinary character of the approach [9].

An important use of system theoretical ideas is found in *biology*. The reason for this is that any kind of organism is an ideal example for a complex system with countless emergent effects. Biochemical mechanisms originating from a large amount of enzyme reaction interactions may be considered as generic multiscale complex system [10]. Moreover, neuro-physiological systems like the human brain illustrate all kinds of phenomena that may arise in the context of complex systems, caused by the complex interconnections of their constituents. The immune system distinguishing between self and non-self is another example for emergent behaviour [11]. Further, the whole biological evolution process is a typical example for dynamical phenomena in complex systems [12]. This is impressively illustrated for evolutionary systems with the help of autonomic, binary, random NK networks. These models mimic the epistatic coupling of genes by connections in boolean networks and show typical complex system behavior like the existing of criticality, which is important in the understanding of the species dependence on random mutations [13]. The problem of self-identification and self-awareness is also a fascinating topic in the *science of cognition* and more general in *philosophy* [14].

A generic complex systems approach is followed in *economics* too, where the idea of positive feedback leads to self-reinforcement mechanisms, explaining well-known economical effects, like product distribution [15, 16]. Also, in *macroeconomics* the mechanisms of financial markets are simulated by evolutionary systems in order to understand the emergence of macroscopic phenomena caused by micro-economical indicators [17]. In general, economics makes use of complex systems in some other contexts too. They are used to describe static corporate structure in *microeconomics* [18] or as a modeling tool in *business administration* [19], where for example the notion of corporate strategy is realized as a system with mutually connected objects corresponding to the various views of value generation, called a balanced scorecard [20]. Some types of evolutionary mechanisms are also introduced and analyzed in economics, where it is argued that changes in corporate structure may be characterized as evolutionary processes which lead among others to self-organization [21].

Social science include a whole system theoretical approach [22] followed by Luhmann's school [23], in which society is considered as a social system consisting of communications between objects [23, 24]. This system theoretical framework is focused on the relations between system and environment and has produced a plethora of sometimes difficult to understand conceptual and theoretical ideas and notions. The specific notion of complexity used in this approach serves as a measure for the system boundary, which is indicated by complexity gradients. Further on, the investigation of artificial societies in the subject of *socionics* relies on the ideas of multi-agents in distributed artificial intelligence [25].

Obviously, systems are used for specification in many *engineering disciplines*, motivated in particular by the usage of *cybernetics*, which is a generic system theoretical discipline [26].

Finally, *computer science* not only make use of system theoretical ideas, but is highly involved in its investigation. *Artificial intelligence* and *artificial life* are concerned with the cognitive aspects of complex systems [27], whereas *software and system engineering* is concerned with specification techniques, conceptual analysis and modeling and simulation of complex systems. Finally, one has to mention the great amount of work concerned with the complex interplay between complex information systems and their surroundings, consisting usually of the system-users and other separated systems.

ToCS and ToCS/SyCL

The *Theory of Complex Systems* (ToCS) as presented in this thesis, is based upon the works of Gerhard Mack [28, 29, 30, 31]. It relies on a universal definition of systems applicable to almost every discipline, while extending this definition by some additional structure concerning locality and possible dynamics. The following preaxiom by G. Mack reflects the central idea of the ToCS approach and builds it semantical core:

Preaxiom (Mack):
The human mind thinks about relations between things or agents.

Based on this statement, ToCS provides a general approach for investigations producing results in form of agent-relation networks. In this sense, the ToCS tries to fill the gap between the highly analytical and quantitative considerations of dynamical systems in the areas of mathematics and physics, and the more structural and qualitative treatment of complex systems in other disciplines (see fig. 1.1). This is demonstrated in particular for the interrelation between ToCS and system-engineering and specification by the parallel development of analytical concepts and of the corresponding implementation of these notions into a executable object-oriented class library, the *System Class Library (SyCL)* [32, 33], for which a graphical user interface was also developed in the recent past.

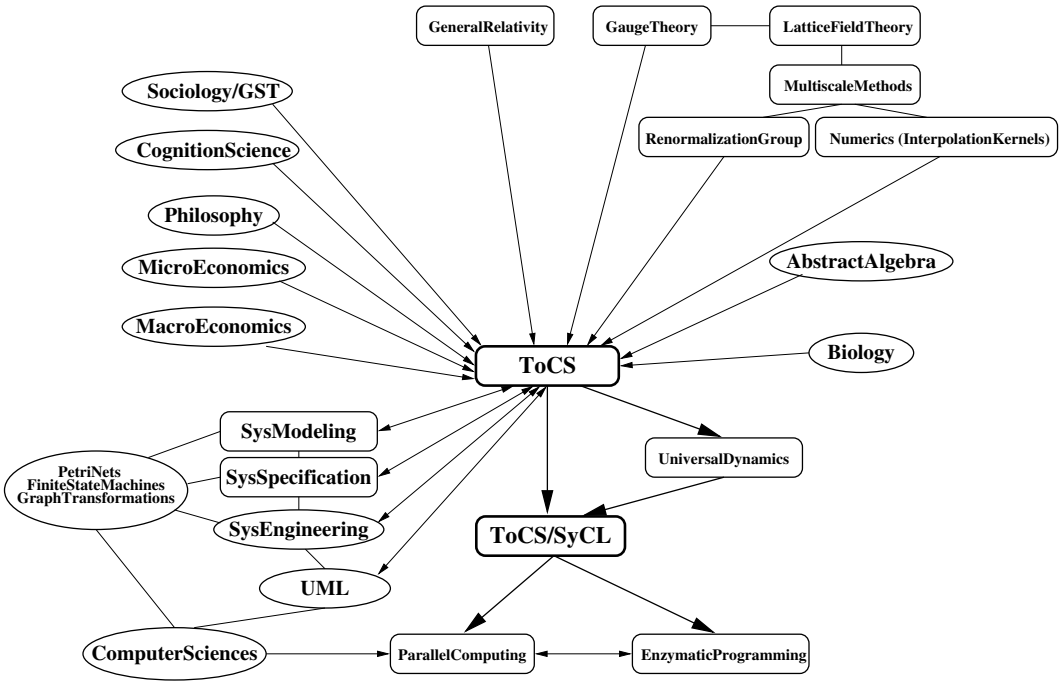


Figure 1.1: The ToCS-based modeling approach and parts of its interdisciplinary context.

The modeling approach specified by ToCS is based essentially on ideas motivated by continuum and lattice gauge theories. The common structure shared by all known gauge theories and general relativity is described in detail in [28, 31, 34]. An impressive illustration of the applicability of gauge theory concepts in less usual contexts is the growing field of *econophysics*, where concepts of finance like the Nobel-price awarded *Black-Sholes equation* are derived as gauge theoretical constructs that are easy to understand in the ToCS framework [35, 36].

However, investigations towards a formal language that allows the description of large scale phenomena in different areas in terms of mathematical systems have been proposed also by other authors [37].

Universality

The interdisciplinary applications of system theoretical concepts motivates its usage as a integrate universal language and modeling tool for the specification and simulation of structural and dynamical aspects in various subjects. As one expects, such an approach must concern mainly structural properties of the investigated model, since it is the structural aspect of a subject of investigation that carries many similarities to semantically different models in other disciplines. The universality of system theoretical concepts is impressively documented in the sociological general system theory (GST) of the Luhmann school, which is basically founded on using ideas from other disciplines, carrying thereby the interdisciplinarity in its core. This property of system theoretical concepts relies mainly on the universal character of its basic definitions (structural and dynamical), which avoid the usage of subject specific properties.

Inabilities of the system theoretical approach

Like every methodology, the advantages of the system theoretical approach depend on the requirements of the problem under consideration. Usually, the strengths of the approach in a given context may be its weaknesses in another scenario. Generally, the inabilities of a methodology separates basically into two questions, namely if the approach is able to describe the fundamental concepts of a given scenario and how far one can go into the details of the involved mechanisms. Obviously, both questions are not independent of each other.

In general the applicability of the ToCS-based approach is restricted by its precise point of view in the sense that it may be infeasible to work out the object-relation structure of a whatsoever situation, a problem that may arise on all scales. On the one side, the system may be not describable as a whole, on the other side, it may be impossible to specialize the general setting of the approach to the requirements of the modeled system. Actually the latter is strongly related to the universality of the theory, which is a great advantage in other contexts.

Chapter 2

THEORY OF COMPLEX SYSTEMS (TOCS)

As mentioned in the last chapter, the notion of systems is useful in many different contexts. To be able to model the structure and behavior of a (complex) system properly, one needs an analytical framework to formulate models of complex systems. The Theory of Complex Systems (ToCS) provides such a framework. Requiring the framework to encompass all basic properties of systems arising in various areas, it should contain a minimum of a priori structure. This motivates an attempt to use mathematical categories as basic tool. They are extended by further properties, when specific scenarios are modeled. The usage of categories assures both, the formal treatment of the system considered, as well as the universal character of the applied mechanisms.

This chapter introduces the basic mathematical entities that are used in ToCS-based framework. It begins with a brief description of the used mathematical tools, like categories, functors and products. Representations of categories are explained and the main representation theorem is proven. A locality principle is defined, turning the category structure into a system structure. Gauge transformations and invariants of categories are explained. Presenting gauge transformations, representations and further constructs in the way it is done here, illustrates the minimal mathematical structure that is needed for their definitions. However, once systems are introduced, it is natural to ask for the types of dynamics that may arise in this framework. Due to the weak a priori structure a system possess, the possible dynamics possess a universal form. After investigating this issue a suggestion for the definition of composite objects is made.[1]

[1] The definitions in this chapter, are as close as possible to the definitions in the ToCS-glossary [38].

2.1 CATEGORIES AND FUNCTORS

The mathematical concept of categories [39, 40, 41] is one of the basic concepts in many formal approaches to complex systems. The reason is, that due to the minimal data a category provides, one is restricted to pure structural formulations. Of course, this extends the applicability of statements to a large class of structures. For the purposes of the theory of complex systems (ToCS) used in this thesis [28, 29, 30], some basic concepts of categories are introduced below. This includes the notion of categories, corresponding structure compatible mappings, called functors, products and examples.

In a sense, a category is a formal network of objects and relations, that connects pairs of objects.

Suppose a set $\mathrm{Ob}(\mathcal{K})$ of **objects** and for each pair in $\mathrm{Ob}(\mathcal{K})$ the set $\mathrm{Ar}(X,Y)$ with

$$X \neq X' \text{ or } Y \neq Y' \;\Rightarrow\; \mathrm{Ar}(X,Y) \cap \mathrm{Ar}(X',Y') = \emptyset \tag{2.1.1}$$

for each $X, Y \in \mathrm{Ob}(\mathcal{K})$ are given. This collection of data is called a **category** $\mathcal{K}$, if there is an associative composition law

$$\mathrm{Ar}(X,Y) \times \mathrm{Ar}(Y,Z) \to \mathrm{Ar}(X,Z) \ ,$$

possessing a neutral element called the **identity** at each object, that is

$$\exists\, \iota_X \in \mathrm{Ar}(X,X) : \ \forall f \in \mathrm{Ar}(X,Y) \ : \ f \circ \iota_X = f = \iota_Y \circ f \quad . \tag{2.1.2}$$

The elements of $\mathrm{Ar}(X,Y)$ are called the **arrows** or **morphisms** of $\mathcal{K}$ from X to Y. We denote arrows $f \in \mathrm{Ar}(X,Y)$ by

$$X \xrightarrow{f} Y$$

and call the disjoint union

$$\mathrm{Ar}(\mathcal{K}) := \dot{\bigcup_{X,Y \in \mathrm{Ob}(\mathcal{K})}} \mathrm{Ar}(X,Y)$$

the arrows of $\mathcal{K}$. Moreover, it is assumed, that there exists mappings α and ω assigning called **source** and **target** respectively,

$$\begin{aligned} \alpha(f) &= X \\ \omega(f) &= Y \quad , \end{aligned}$$

for an arrow $X \xrightarrow{f} Y$.

It is possible to generalize the definition of $\mathrm{Ob}(\mathcal{K})$ and $\mathrm{Ar}(\mathcal{K})$ to be mathematical classes instead of sets. For the purposes of this thesis, the definition is in fact that of *small* categories.

The trivial category $\mathrm{Ob}(\mathcal{K}) = \emptyset$, is denoted by $\mathcal{K} = 0$, and referred to as the **archetype 0**. Similarly, a category consisting of n isolated objects together with their n identities is called the **archetype n**.

Examples

(i) The set of small sets, with set theoretical maps considered as arrows, forms a category denoted by **Set**.
(ii) Considering (small) groups as objects and group-homomorphisms as arrows, the set of groups becomes a category denoted by **Grp**.

An arrow $f^{-1} \in \mathrm{Ar}(Y, X)$ is called the **inverse** of $f \in \mathrm{Ar}(X, Y)$, if

$$\begin{aligned} f \circ f^{-1} &= \iota_Y \\ f^{-1} \circ f &= \iota_X \ . \end{aligned}$$

Categories may carry additional structure, relating to each arrow between two objects an arrow in the opposite direction, called its adjoint arrow. More formally:

A category $\mathcal{K}$ is called a ***-category**, if there exists an **adjoint-operation** * on $\mathrm{Ar}(\mathcal{K})$, defined by the properties

$$\begin{aligned} &(i) && \mathrm{Ar}(X, Y) \xrightarrow{*} \mathrm{Ar}(Y, X) \\ &(ii) && {\iota_X}^* = \iota_X \\ &(iii) && f^{**} = f \\ &(iv) && (g \circ f)^* = f^* \circ g^* \ . \end{aligned}$$

Structure-preserving maps of categories are called functors.

Given two categories $\mathcal{K}$ and $\mathcal{K}'$, a **covariant functor** F is a correspondence, which assigns to each object $X \in \mathrm{Ob}(\mathcal{K})$ in $\mathcal{K}$ an object $F(X) \in \mathrm{Ob}(\mathcal{K}')$ in $\mathcal{K}'$ and to each arrow $f \in \mathrm{Ar}(X, Y)$ an arrow $F(f) \in \mathrm{Ar}(F(X), F(Y))$, fulfilling

$$F(\iota_X) = \iota_{F(X)} \tag{2.1.3}$$

$$F(g \circ f) = F(g) \circ F(f) \ . \tag{2.1.4}$$

An injective and surjective (faithful) functor is called an **equivalence** (or an **isomorphism**), the corresponding categories being referred to as **isomorphic**.

Examples

(i) Considering each group as a set and each group-homomorphism as a map between sets, a functor from **Grp** to **Set** is trivially defined(*forgetful functor*).
(ii) Consider the category of manifolds **Mf**. The arrows between two objects will be homotopy classes of maps. Assigning to each n-dimensional manifold its n^{th} homology group H_n, we get a covariant functor between **Mf** and the category of groups **Grp**.

Similarly to the definition above, a **contravariant functor** is defined by reversing the direction of the arrows $F(f) \in \mathrm{Ar}(F(Y), F(X))$ and consequently $F(g \circ f) = F(f) \circ F(g)$. Usually the term *functor* is used to refer to covariant functors.

Usually, functors are assumed to commute with $*$-operations, that is

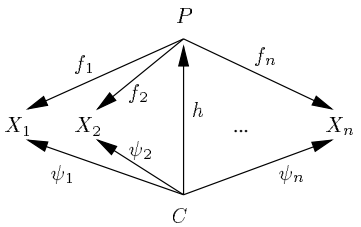
$$F(f^*) = F(f)^* \quad .$$

Products

Some constructions use the concepts of products of objects in a category. Consider a family of objects $X_i \in \mathrm{Ob}(\mathcal{K})$. Then $(P, \{f_i\})$, with $P \in \mathrm{Ob}(\mathcal{K})$ and $f_i \in \mathrm{Ar}(P, X_i)$, is called a **product** of the X_i, if for every $C \in \mathrm{Ob}(\mathcal{K})$ and $\psi_i \in \mathrm{Ar}(C, X_i)$ there exists a unique arrow $h \in \mathrm{Ar}(C, P)$, such that

$$\psi_i = f_i \circ h \ , \ \forall i,$$

that is, making the following diagram commutative:



Coproducts are defined in complete analogy by reverting all arrows.

Considering for example the category of sets and specifying the f_i to be the projections on the X_i this product becomes the ordinary direct product of sets.
To generalize this notion to all kinds of products we consider the following scheme:
First, define an object P to be **universal attracting**, if there is a unique arrow from each object to P. Conversely an object P is defined to be **universal repelling**, if there is a unique arrow from P to

each object in the category.

Now take the above family of objects $\{X_i\}$ and define a new category $\mathcal{K}'$ whose objects are families of arrows

$$\{f_i : Y \to X_i\}_{i \in I}$$

and whose arrows between objects

$$\{f_i : Y \to X_i\} \text{ and } \{f_i' : Y' \to X_i\}$$

consists of arrows $h : Y' \to Y$ in $\mathcal{K}$, such that

$$f_i' = f_i \circ h \ .$$

A product of the X_i is then defined to be a universal attracting object in $\mathcal{K}'$. In analogy, by reverting the directions of all arrows in the above construction, a coproduct of the X_i is defined to be a universal repelling object in the category $\mathcal{K}''$ consisting of families

$$\{f_i : X_i \to Y\} \ .$$

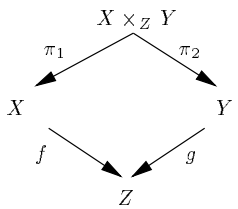
Note however, that the arrows h depend heavily on the choice of the f_i.

Examples

(i) Consider the fibre product of objects X and Y over Z. We define a new category $\mathcal{K}_Z$ whose objects are the arrows of $\mathcal{K}$ and whose arrows between two objects

$$\{f : X \to Z\} \text{ and } \{g : Y \to Z\}$$

are arrows $\phi : X \to Y$, such that $f = g \circ \phi$. A product of objects in $\mathcal{K}_Z$ is called a fibre product of the corresponding arrows in $\mathcal{K}$ and is denoted by $X \times_Z Y$. The commutative diagram illustrates



this situation.

(ii) The tensor product of commutative rings can be defined in this way. Let $\mathbf{Rg}$ be the category of commutative rings with ring-homomorphisms as arrows. Given objects X and Y there exist natural ring-homomorphisms $X \to X \otimes Y$ and $Y \to X \otimes Y$ from these objects into their tensor product

$$\begin{aligned} X \ni x &\mapsto x \otimes \mathbf{1} \\ Y \ni y &\mapsto \mathbf{1} \otimes y \end{aligned}$$

making the tensor product $X \otimes Y$ a coproduct in the category $\mathbf{Rg}$ in the sense defined above.

2.2 REPRESENTATIONS

Categories are abstract constructions reflecting in mathematical fashion the structure of a collection of related entities. To extract interpretational information from a category, one makes use of representations. A representation maps the category onto a well-known structure (category). More formally:

> A **representation** of a category $\mathcal{K}$ onto a category $\mathcal{K}'$ is a surjective covariant functor ρ from $\mathcal{K}$ to $\mathcal{K}'$.

It is assumed that representations preserve additional structure of the represented category. The following theorem enables one to treat objects as sets and arrows as maps between them, due to a constructive representation of the given category onto **Set**.

> **Main Representation Theorem:**
> *Every small category $\mathcal{K}$ admits a faithful representation on a subcategory $\mathcal{C}$ of the category of sets* **Set**.

To prove this, for an arbitrary small category $\mathcal{K}$ its so called **communication network** is constructed out of the given data. The main representation theorem follows from this construction as a corrolary.[2]

> **Communication Network Construction:**
> *Given a category $\mathcal{K}$ there exists for every $X \in Ob(\mathcal{K})$ sets A_X (input space) and Ω_X (output space) and a correspondence $\tilde{}$ which assigns to each X an invertible map*
>
> $$\tilde{X} : A_X \to \Omega_X \tag{2.2.1}$$
>
> *and to each $f \in Ar(X, Y)$ a map*
>
> $$\tilde{f} : \Omega_X \to A_Y \tag{2.2.2}$$
>
> *with the properties*
>
> $$\tilde{X} \circ \tilde{\iota}_X = id_{\Omega_X} \quad and \quad \tilde{\iota}_X \circ \tilde{X} = id_{A_X} \tag{2.2.3}$$
>
> $$\widetilde{g \circ f} = \tilde{g} \circ \tilde{Y} \circ \tilde{f} : \Omega_X \to A_Z \, . \tag{2.2.4}$$

[2] The proof of both is given in appendix D.

Due to the *Main Representation Theorem* categories may be treated almost like lattice gauge theories [42], where objects become vector spaces, in which the fields take their values. The arrows become (linear) parallel transporters along edges of the lattice belonging to some gauge group. The main difference to this picture, lies in the fact, that the sets Ω_X and the maps $\rho(f)$ of the represented category will not be linear in general. This illustrates also the generalization aspect, originating from the categorical notion.

Note, that the representation constructed in the proof of the *Main Representation Theorem* is not unique. Performing nontrivial gauge transformations either, of the original category, or of the represented category one gets different, but isomorphic representations of the original category.

Usually, there will be also many not isomorphic representations of a given category $\mathcal{K}$. Let all representations ρ of $\mathcal{K}$ be labelled by an index α, where ρ_0 denotes the identity representation of $\mathcal{K}$ into $\mathcal{K}$. We call the set $\{\rho_\alpha(X) \ : \forall\alpha\}$ the **concepts** of X, whereas the representation images of the objects and arrows of $\mathcal{K}$ in $\rho_\alpha[\mathcal{K}]$ are called their $\boldsymbol{\rho_\alpha}$ **-meaning** respectively.

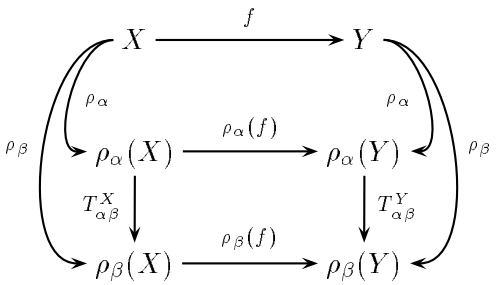
Considering the concepts of X as a collection of objects, one may ask, if there do exist arrows between them, turning this collection and thereby different representations of $\mathcal{K}$ into a nontrivial category.
Arrows T^X between different meanings of an object X of $\mathcal{K}$

$$T^X_{\alpha\beta} \ : \ \rho_\alpha(X) \to \rho_\beta(X)$$

will be called **intertwiners**, if they satisfy the following relation [3]

$$T^Y_{\alpha\beta}\,\rho_\alpha(f) = \rho_\beta(f)\,T^X_{\alpha\beta}\ , \tag{2.2.5}$$

for every arrow $f : X \to Y$. This definition generalizes the familiar notion of intertwiners to categories. The several representations of a given category are related as shown in the following commutative diagram:



[3] For two identical representation categories $\rho_\alpha(\mathcal{K})$ and $\rho_\beta(\mathcal{K})$ this relation makes the $T_{\alpha\beta}$ become natural transformations.

As shown in later sections, a special case of intertwining relations occurs for interfaces of composite objects. If we consider the identity representation $\rho_\alpha := \rho_0$, the existence of intertwiners $T^X_{0\beta}$ enables a non-trivial gluing of the original category $\mathcal{K}$ and its representation $\rho[\mathcal{K}]$ to a larger one. Its arrows consist of the sets $\mathrm{Ar}(\mathcal{K})$ and $\mathrm{Ar}(\rho(\mathcal{K}))$ and the components $\rho_\beta(X)$ of the representation ρ_β, the later being arrows themselves. The intertwiner-relation (2.2.5) then becomes

$$\rho^Y f \;=\; \rho(f)\rho^X \quad , \tag{2.2.6}$$

where we have set $\rho := \rho_\beta$ and $\rho^X := T^X_{0\beta}$.
The relation (2.2.6) is considered as a constraint relation for the " gluing ", assuming it in some sense to be unfrustrated.

2.3 SYSTEMS

Up to this point, the underlying structure was assumed to be a category. To model a given scenario, the basic agents of the scenario will be defined to be the objects in an appropriate category, while the interrelations of the entities will become the arrows of the category. Lattice gauge theories provide premium examples for this [42]. To each vertex of the lattice its state space (where the fields take their values) is assigned as object. To each link the gauge parallel transporters along this link define the arrows between these objects. This situation can be naturally extended to ordinary gauge theories defined in the continuum. The main difference lies in the fact, that in models of lattice gauge theory, there is a feasible definition of a finite and discrete neighborhood to each object, originating from the neighbors of the associated vertex in the lattice.

Moreover, to proceed to the formulation of dynamics in a systemic approach, it is necessary to clarify how an evolution of a structure may occur in this framework. In the examples above, where space time points (or their associated state space) are represented as objects of a category, the question arise, how an equation of motion for states of the system could be formulated in this general setting. Since all known fundamental physical laws are of local character, it seems instructive, to define a locality concept as a basic notion. In ToCS, one may define for each object a distinguished collection of other objects in the category, which determines its neighborhood. This is realized by labelling some arrows as fundamental arrows.

Given a subset $\mathrm{Fund}(\mathcal{K}) \subset \mathrm{Ar}(\mathcal{K})$ of arrows, that satisfy the condition

$$\forall f \in \mathrm{Ar}(\mathcal{K}) \;\; , \exists b_\alpha \in \mathrm{Fund}(\mathcal{K}) \;\; : \;\; f = \prod_\alpha b_\alpha \;\; ,$$

where the product is with respect to the composition law. We call $\mathrm{Fund}(\mathcal{K})$ a set of **fundamental arrows** or **links** for $\mathcal{K}$. We set

$$\mathrm{Fund}(X,Y) := \mathrm{Fund}(\mathcal{K}) \cap \mathrm{Ar}(X,Y) \;\; ,$$

for the fundamental arrows between two objects X and Y.

The set $\mathrm{Fund}(\mathcal{K})$ generates the set of arrows $\mathrm{Ar}(\mathcal{K})$ with respect to the composition law. Since the identities in a category do not contribute to the decomposition of an arrow into links, one may restrict the definition of links by requiring, that no identity is included in $\mathrm{Fund}(\mathcal{K})$. Until stated differently, this is assumed in the following.

The **radical** $\mathrm{Rad}(X)$ is associated to an object X, it is defined as follows

$$\mathrm{Rad}(X) := \{\ (X,b)\ :\ b \in \bigcup_Z \mathrm{Fund}(Z,X)\ \} \quad . \tag{2.3.1}$$

Moreover, one may define the **neighborhood** $\mathrm{Nbh}(U)$ of an object, link, or subsystem U to be the set of objects and links, which can be reached by traversing (from its target to its source) at most one link, external to the considered object, link, or subsystem. For an object, this definition is identical to the radical of the object. A link is included in the neighborhood if its source or target may be reached by a link.

With this data a system is defined as follows:

> A (general) **system** $\mathcal{S}$ is a pair $(\mathcal{K}, \mathrm{Fund}(\mathcal{K}))$, where $\mathcal{K}$ is a category and $\mathrm{Fund}(\mathcal{K})$ is a set of fundamental arrows.

Obviously, every category may be considered as a system in a trivial way, with fundamental arrows being the whole of $\mathrm{Ar}(\mathcal{K})$ (without the identities). However, in many cases further restrictions for the fundamental arrows will be assumed, like the cardinality of the neighborhood of an object being finite and/or identical for all objects.

The category $\mathcal{K}$ may carry an $*$-operation defined on $\mathrm{Ar}(\mathcal{K})$. Of course, this structure is inherited by the system. However, if this operation can be consistently restricted to $\mathrm{Fund}(\mathcal{K})$, that is, if the adjoint of a link exists and is a link itself, we call the system a **complete system**. Conversely, if a complete system is given by an $*$-operation on $\mathrm{Fund}(\mathcal{K})$ the underlying category becomes automatically an $*$-category by extending the $*$-operation on $\mathrm{Ar}(\mathcal{K})$ via composition of fundamental arrows. It is important to note, that for some $*$-systems, which are not complete, the definition of fundamental arrows is modified. There, one requires every arrow to be decomposable into a product of links or adjoints of links.

Paths and constraints

Starting with a system $\mathcal{S}$, its fundamental arrows may be mapped canonically into $\mathrm{Ob}(\mathcal{K}) \times \mathrm{Ob}(\mathcal{K})$ via their sources α and their targets ω

$$\begin{aligned} B : \mathrm{Fund}(\mathcal{K}) &\rightarrow \mathrm{Ob}(\mathcal{K}) \times \mathrm{Ob}(\mathcal{K}) \\ b &\mapsto (\alpha(b), \omega(b)) \ . \end{aligned}$$

If B is injective, this defines a directed graph $\Gamma(\mathcal{S})$, consisting of vertices $X \in \mathrm{Ob}(\mathcal{K})$ and edges, which are given by $B[\mathrm{Fund}(\mathcal{K})]$. In this case, one may equivalently define the system to be the pair $(\mathcal{K}, \Gamma(\mathcal{S}))$.

Conversely given the directed graph $\Gamma(\mathcal{S})$ one may construct a new category $\mathcal{K}'$ and thereby a new system $\mathcal{S}'(\Gamma)$. Objects are defined to be the vertices of the graph, arrows f are defined to be **paths**, that is, formal series of successive edges b_i

$$f := b_n \, ... \, b_2 \, b_1 \ .$$

The product of paths $\circ'$ is defined by juxtaposition, that is,

$$f \circ' g := b_n \, ... \, b_2 \, b_1 \, d_l \, ... \, d_2 \, d_1 \ \ ,$$

for $f = b_n ... b_2 b_1$ and $g = d_l ... d_2 d_1$. Adding for each object its identity we get the category $(\mathcal{K}', \circ')$, with objects identical to that of the original category $\mathcal{K}$. Defining the edges of the graph to be the fundamental arrows, from and from $(\mathcal{K}', \circ')$ the system $\mathcal{S}'(\Gamma)$ is constructed.

The categories $\mathcal{K}$ and $\mathcal{K}'$ and thereby the systems $\mathcal{S}$ and $\mathcal{S}'(\Gamma)$ need not be identical at all, since the reconstructed set of arrows $\mathrm{Ar}(\mathcal{K}')$ will be in general much greater then $\mathrm{Ar}(\mathcal{K})$. This is due to the fact, that the decomposition of a given arrow into links in the original category $\mathcal{K}$ may not be unique, whereas the one in $\mathcal{K}'$ is so by definition. For the reconstructed category and system to be identical to the original data, a **generating set of constraints** must be imposed for products of fundamental arrows that will identify the paths, that correspond to identical arrows in the original category. If the original graph was given by a system, one expects the classes of identified paths to possess a composition structure, that is, the system should be given by classes of identified paths as its arrows. However, in general this needs not to be the case. A system, with imposed constraints will be called a **constrained system**. For physical systems such relations will be given by the constraints of the theory.

Since we assume the dynamical laws to be local, the most important constraints will arise from canonical constructions, with as minimal as possible links, that is, from local constraints. Examples of such relations for complete systems are the following

$$\prod b^* \circ b = \iota_X \ ,$$

where the product is over a subset of fundamental arrows starting at the same object X.[4] Examples of constraints for general systems are

$$\prod_{\Delta_X} \Delta_X = \iota_X \ ,$$

[4] As shown in chapter 3, the Gauss law in classical electrodynamics is implemented in the ToCS-based framework by a relation of this type.

where Δ_X is a triangle at X, that is, an arrow of the local gauge semi group,[5] which can be decomposed into links in the following manner

$$\Delta_X = b_3 \circ b_2 \circ b_1 \ ,$$

with $\alpha(b_1) = \omega(b_3) = X$ and $\omega(b_2) = \alpha(b_3) = Y$ and $\omega(b_1) = \alpha(b_2)$. Finally, this type of relation is slightly generalized to

$$g_X = \iota_X \ ,$$

for elements g_X in a subset of the local gauge semi group G_X.

Systems are called **connected**, if every object in the system is source or target of at least one link. Obviously for a connected system the underlying category is connected. Bidirectional connectedness is defined in analogy.

Functors of systems are defined to be functors of the associated categories. Usually, they will not preserve the system structure, that is, the (multi-edged) graphs of the systems will not be homomorphically mapped into each other. However, in many cases functors are assumed to commute with a given $*$-operation on the systems

$$F(f^*) = F(f)^* \ .$$

Clearly all further notions like representations, or products defined for categories may be applied on systems without any modifications. Functors are called *system structure preserving* if they map links into links preserving thereby the associated graphs.
In this sense an isomorphism F of systems is an equivalence of the corresponding categories, which preserve the system structure 'in both directions', that is

$$F[\,\mathrm{Fund}(\mathcal{K})\,] = \mathrm{Fund}(F[\mathcal{K}]) \ .$$

Layered systems

Suppose a decomposition of the set of objects $\mathrm{Ob}(\mathcal{K})$ of a system $\mathcal{S}$ into disjunct subsets $\mathrm{Ob}(\mathcal{K}_l)$ is given. Then, considering the corresponding sets of arrows $\mathrm{Ar}(\mathcal{K}_l)$, one gets subcategories $\mathcal{K}_l$. However, it is not necessary for $\mathcal{K}_l$ to inherit a system structure by restricting $\mathrm{Fund}(\mathcal{K})$ to $\mathrm{Ar}(\mathcal{K}_l)$. Therefore the following definition is useful:

> A **layered system** $\mathcal{S}$ is a system that contains n disjunct subsystems $\mathcal{S}_l$, with a natural $l = 1, ..., n$ fulfilling
>
> $$\begin{aligned} \mathrm{Ob}(\mathcal{S}_l) \cap \mathrm{Ob}(\mathcal{S}_{l'}) &= \emptyset \ , \quad \forall l \neq l' \\ \mathrm{Fund}(\mathcal{S}_l) \cap \mathrm{Fund}(\mathcal{S}_{l'}) &= \emptyset \ , \quad \forall l \neq l' \ . \end{aligned}$$

[5] The local gauge semi group is the set of arrows with identical source and target, called loops.

The remaining links

$$I_l := \text{Fund}(\mathcal{S}) \setminus \bigcup_l \text{Fund}(\mathcal{S}_l) \quad ,$$

are links, that relate following layers $\mathcal{S}_l$ and $\mathcal{S}_{l\pm1}$. They are called **inter-layer-links**.

In essence, layers are used to distinguish and label some sets of arrows by their source and target in different layers.

Layered systems are used for example in the definition of dynamics, or in P/T-systems, described in chapter 3.

Completing a system

At this point we remark an assumption, which is important for the implementation of some types of dynamical laws. Given a complete system $\mathcal{S}$, from its associated graph $\Gamma(\mathcal{S})$ one may define a new graph Γ' by declaring some adjoint links to be absent and reconstruct thereby a different system $\mathcal{S}'$ out of Γ'. This system will not be complete in general, since the $*$-operation of the original system is now defined only on a subset of Fund$(\mathcal{K}')$.[6] Nevertheless, since the missing adjoints are included in the underlying category,the system $\mathcal{S}'$ can be completed in a unique manner by extending the fundamental arrows to close under the $*$-operation. For all practical purposes, it is claimed, that the considered systems arise from complete ones, that is, the following assumption is stated:

Assumption:
Every considered system can be extended uniquely to a complete system.

Although this assumption seems rather strong, it will be fulfilled canonically in many cases considered in the next chapters.

The collection of all statements that makes sense for arbitrary systems is referred to as the *language of thought*.

Example

Consider the logical archetype **LAT** [31]: It is defined as a complete system, consisting of objects $\{T, F\}$ and links

$$\begin{aligned} e &: T \to F \\ e^* &: F \to T \\ o &: F \to F \end{aligned}$$

[6] In this case, arrows are decomposed into links and adjoint of links, which may be not links themselves.

and the identities ι_T and ι_F. They are assumed to satisfy the relations

$$\begin{aligned} o^* &= o \quad \text{(self-adjoint)} \\ e^* \circ e &= \iota_T \\ e \circ e^* &= \iota_F \\ o \circ o &= o \ , \end{aligned}$$

from which the unitarity of e follows

$$e^{-1} = e^* \ .$$

The local gauge semi groups G_F and G_T are isomorphic

$$\begin{aligned} \mathrm{G}_F &= \{\iota_F, o\} \\ \mathrm{G}_T &= \{\iota_T, e^* \circ o \circ e\} \ . \end{aligned}$$

The system **LAT** serves for logical representations of systems. For this, the objects T and F are interpreted as *truth* and *false*, the relations e and e^* are interpreted as *negations*, o may be thought as *adding an arbitrary proposition to a given false one*. This allows one to assign truth values to systems. Given a representation of a system onto **LAT**, the objects of the system are interpreted as propositions, while its arrows $f : X \to Y$ (except the identity) become relations "X *excludes* Y".

2.4 GAUGE TRANSFORMATIONS

As known from more conventional theoretical frameworks, the definition of a physical system often involves redundancies in the form of unobservable degrees of freedom, which are related to the existence of gauge transformations [43, 44]. Due to these redundancies, gauge orbits of the original system must be divided out and require thereby the redefinition of the physical system as a class of gauge equivalent systems.

Such a lack of uniqueness appears in the general setting considered here, too. It relies on the fact, that the definition of a category enables the canonical construction of a collection of categories. The functors associated to the transformations into these categories are called gauge transformations.

More precisely, we introduce for each object $X \in \mathrm{Ob}(\mathcal{K})$ its **local gauge semi group** $\mathrm{G}_X := \mathrm{Ar}(X, X) \subset \mathrm{Ar}(\mathcal{K})$. The elements g_X of G_X are arrows, possessing identical source and target object X, and are called the **loops** at X.

Suppose now a section is given in the bundle

$$\bigcup_{X \in \mathrm{Ob}(\mathcal{K})} \{X\} \times \mathrm{G}_X \ ,$$

that is, a (unique) loop g_X at each object of the given category. From this, we construct an operation $'$ on $\mathcal{K}$ which preserves the objects of $\mathcal{K}$ and acts on the arrows of $\mathcal{K}$ by

$$\begin{aligned} \mathrm{Ar}(X,Y) &\rightarrow \mathrm{Ar}(X,Y) \\ f &\mapsto f' \ , \end{aligned}$$

fulfilling the following condition for all $X, Y \in \mathrm{Ob}(\mathcal{K})$

$$g_Y \circ f = f' \circ g_X \ . \tag{2.4.1}$$

Since this implies for $h \in \mathrm{Ar}(X,Y)$ and $f \in \mathrm{Ar}(Y,Z)$ the relation

$$(f \circ h)' \circ g_X = (f' \circ h') \circ g_X \ ,$$

we notice, that it is always possible to define the operation in a way, that it becomes functorial. A functor, which operates on $\mathcal{K}$ in the above manner, is called a **gauge transformation** of the category.
Assuming that with $g_X \in \mathrm{G}_X$ there exists the inverse ${g_X}^{-1} \in \mathrm{G}_X$ of each loop, G_X becomes a group and (2.4.1) takes the more familiar form

$$f' = g_Y \circ f \circ {g_X}^{-1} \ , \tag{2.4.2}$$

in analogy to the homogeneous transformation law for parallel transporters in ordinary gauge theories. To reproduce the known gauge theories of elementary particles, categories with gauge groups isomorphic to the corresponding SO(n) and SU(n) must be involved. Usually they may be achieved by defining the arrows of the category to be the parallel transporters of the theory, as explained for classical electrodynamics in the next chapter. Unitarity in this context means, that the adjoint of an arrow is its inverse [34].

Natural transformations

It is possible to state the definition of gauge transformations in a somewhat more structural fashion. For this, suppose two functors F and G are given, which map the category $\mathcal{K}$ into the same category $\mathcal{K}'$. A **natural transformation** of F into G is an operation α, which associates to each object X of $\mathcal{K}$ an morphism $\alpha^X \in \mathrm{Ar}(F(X), G(X))$, making the following diagram commutative:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \alpha^X}\downarrow & & \downarrow{\scriptstyle \alpha^Y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array}$$

The α^X is called the component of α at X. In this sense, natural transformations can be considered as arrows of functors.

Gauge transformations arise via specific natural transformations: Setting $F(f) = f$ (identity functor), $G(f) = f'$ (gauge transformation) and $\alpha^X := g_X$ for a given gauge transformation, the relation (2.4.1) yields the following result:

Result:
Every gauge transformation admits a well-defined natural transformation to the identity functor.

Frustration and invariants

Of major importance when investigating the complexity of a given category is the number of different arrows between two objects. A category $\mathcal{K}$ will be called **unfrustrated** if there is at most one arrow between arbitrary objects X and Y and **frustrated** otherwise. By definition, unfrustrated categories possess trivial local gauge semi groups $\mathrm{G}_X = \{\iota_X\},\ \forall X$ and therefore no nontrivial gauge transformations occur in these cases. Triangulations of flat, nonsingular spaces are examples of unfrustrated categories. Further, every commutative diagram may be considered as a unfrustrated category.

We will now proceed to the definition of functions that remain invariant under an appropriate local action of the arrows of the category. This applies in the case of subcategories of **Set**, the generalization of this concept to arbitrary categories is considered in what follows. For this, assume $\mathcal{K}$ is a subcategory of **Set**, so that the objects Ω_X become sets.[7] Suppose now a family of functions $(\chi_X)_{X\in \mathrm{Ob}(\mathcal{K})}$ from the n^{th} Cartesian power of Ω_X into a set M is given,

$$\chi_X : {\Omega_X}^n \mapsto M$$

we call this family an **invariant field**, if its values remain constant under the action of arrows

$$\chi_X(\xi_1, ..., \xi_n) = \chi_{\omega(f)}(f(\xi_1), ..., f(\xi_n)) \ , \tag{2.4.3}$$

where $\omega(f)$ is the **target** of $f \in \mathrm{Ar}(X, Y)$, that is, $\omega(f) = Y$.

The above definition is generalized to the case of arbitrary categories by the following consideration. Given an invariant field $(\chi_X)_X$, we construct a new category $\mathcal{K}'$, with objects

[7] The main representation theorem ensures, that for any given small category, there exists an isomorphic subcategory of **Set**.

$\mathrm{Ob}(\mathcal{K}')$ being the functions χ_X themselves. The realizing functor F is defined canonically as follows:

$$\begin{array}{ll} (i) & F(\Omega_X) := \chi_X \\ (ii) & F(f) := \chi_{\omega(f)} \circ f^{\times n} \text{ for } f \in \mathrm{Ar}(X,Y) \\ (iii) & F(f) \circ' F(g) := \chi_{\omega(f \circ g)} \circ (f \circ g)^{\times n} \ . \end{array}$$

Due to the definition of invariant fields, this category is unfrustrated. From this, we define for an arbitrary category $\mathcal{K}$ an **invariant** to be a functor F of $\mathcal{K}$ into a unfrustrated category. Obviously, invariants are gauge invariant, that is $F(f') = F(f)$ for a gauge transformation $f \mapsto f'$.

2.5 DYNAMICS OF SYSTEMS

The objects and links in a system represent the structure of a modeled scenario. However, as time evolves, the structure of the system may change, by means of modifications in the set of objects and their relating arrows. The question arises, what the possible changes in a system may be. Since the data provided by a system are very restricted, there are only a few possibilities to operate generically on a system. First, one may change the set of objects, by adding, or deleting one or more objects (growth of the system). This is possible canonically, since to delete an object one does not need further information, while in copying an object, one makes use only of the original one. Second, one may change the set of fundamental arrows, by declaring arrows to be fundamental (motion in the system). At least, one may modify the composition law, by traversing an additional loop at each object, where a composition is performed.

The *dynamics* of a system will be defined as a rule, which provides a family of systems, based on a combination of the described system modifications, the corresponding family of systems will then be called a *drama*.

However, if the set of objects is not affected by the dynamics, the underlying category remains invariant. In these cases, dynamics is dynamics of the system structure, that is, dynamics of the local structure at an object. Stated differently, this means, that the neighborhood of an object has changed. From an operational point of view, this may be considered as motion of the object. Conventionally this is expressed by a function, assigning to each time a point in (a topological) space, but this may be as well considered as a time dependent relation between a hypothetical particle and this space-point. Motion of the particle then simply means, that this relation changes dynamically in time.

Dynamics and Enzymes

Although some constructions may be easily extended to the continuum, in the following time is assumed to be discrete. The time step is denoted by τ.

More precisely, the considerations above motivate the following definition:

A **drama** is a layered system $(\mathcal{S}_t)_t$, whose inter-layer-links are unitary.

The unitary inter-layer-links connect an object with its descendants.

The crucial point in the definition of dynamics is, that the layer $\mathcal{S}_t$ of a drama should be given by some (predecessors) layers $\mathcal{S}_{t-\tau}, ..., \mathcal{S}_{t-n\tau}$, that is, by the action of a specific dynamics law. A **dynamics law** is a mapping

$$t \mapsto \mathcal{S}_t \quad ,$$

which provides the layers of a drama. However, the rule that operates on a given system is restricted to be of a certain order.

A **n-th order dynamics law** is a rule, that provides a map H_t

$$H_t \ : \ (\mathcal{S}_{t-\tau}, ..., \mathcal{S}_{t-n\tau}) \ \mapsto \ \mathcal{S}_t \quad ,$$

for the layers in a drama $(\mathcal{S}_t)_t$.

The dynamics is called **autonomous**, if

$$H_t = H_{t+\tau} \quad ,$$

for all times t, that is, if the rule is time-independent.

The most important types of dynamics are local dynamics laws.

A **n-th order local dynamics law** is a n-th oder dynamics law, fulfilling the following properties:

1. Every object X' in $\mathcal{S}_{t+\tau}$ is descendent of at least one object X in $\mathcal{S}_t$.
2. Every link b in $\mathcal{S}_{t+\tau}$ is given by a product

$$b = \prod_{c_i \in \mathrm{Nbh}^n(b)} c_i(b) \quad , \tag{2.5.1}$$

of links c_i in

$$\mathrm{Nbh}^n(b) := \bigcup_{k=1}^{n} \mathrm{Nbh}_{t-k\tau}(b) \quad . \tag{2.5.2}$$

The set $\mathrm{Nbh}^n(b)$ contains the union of the neighborhoods of the link b at n earlier times.[8] If we assume, that no change occur in the set of objects, the underlying category remains the same through the whole evolution of the system. As a consequence, the set of arrows remains unchanged, but the decomposition of a given arrow into fundamental ones varies in time. This means

$$\mathcal{K}_{t+\tau} \;=\; \mathcal{K}_t \quad , \tag{2.5.3}$$

for every t, where we have set $\mathcal{S}_t \;=\; (\mathcal{K}_t, \mathrm{Fund}(\mathcal{K}_t))$ for the underlying category. Since (2.5.3) implies that the set of arrows remains invariant, the set of fundamental arrows at a given time can be embedded in every set $\mathrm{Ar}(\mathcal{K}_t)$, that is

$$\mathrm{Fund}(\mathcal{K}_{t+k\tau}) \hookrightarrow \mathrm{Ar}(\mathcal{K}_t) \qquad \forall t \quad , \tag{2.5.4}$$

where k is an integer $k \in \mathbb{Z}$. This makes the decomposition (2.5.1) of the link b well-defined.

The composition law at each object may be modified by

$$f \circ g \to f \circ l \circ g \quad ,$$

where l is a loop. This is not further considered here.

An autonomous, local dynamics is called a **universal dynamics**, if it is applicable to every system. In agreement with the implementation of the theoretical framework into SyCL (see appendix A), the following definition is useful:

> A rule that operates on a system to produce a local dynamics law, is called an **enzyme**. The enzyme is called a **universal enzyme**, if the law is a universal dynamics.[9]

The general dynamics of a system is thought to be given by a sequence of enzyme actions. Examples for dynamics laws, originating from enzymatic actions are given at end of this section and in the next chapter.

As mentioned in the introduction to this section, the minimal data of a system restrict the possible dynamics considerably. The declaration of an arrow to be fundamental a time step later, is called **motion**. A simple example is to declare

$$b_i \circ b_j \in \mathrm{Fund}(\mathcal{S}_{t+\tau}) \quad , \quad \forall b_i, b_j \in \mathrm{Fund}(\mathcal{S}_t) \quad .$$

[8] In general, this is the neighborhood of a path.

[9] In its implementation, the locality of the enzymatic action is realized by its inherited object-properties, that means that it possesses a local neighborhood on which it acts.

However, the question arises, if the arrows b_i and b_j remain fundamental or not. In **simple motion**, the following is assumed

$$\begin{aligned} b_i \circ b_j &\in \text{Fund}(\mathcal{S}_{t+\tau}) \\ b_i, b_j &\notin \text{Fund}(\mathcal{S}_{t+\tau}) \quad , \end{aligned}$$

whereas in any other case, the enzyme will be called **motion with trace**. A frequently used variant is

$$\begin{aligned} b_i, b_i \circ b_j &\in \text{Fund}(\mathcal{S}_{t+\tau}) \\ b_j &\notin \text{Fund}(\mathcal{S}_{t+\tau}) \quad . \end{aligned}$$

Based on elementary actions one may construct more complex ones by building action sequences. The basic enzymes used for this purpose are called **fundamental enzymes**.

As demonstrated in chapter 3 for classical electrodynamics, a n-th order dynamics may be reformulated as a first-order dynamics by extending the set of arrows. This corresponds to the definition of canonical pairs in canonical formulations.

Example

To illustrate the action of a sequence of fundamental enzymes, the *SplitFork* enzyme is shown in fig. 2.1.

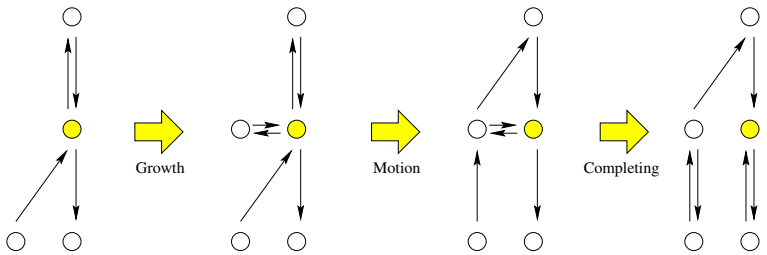


Figure 2.1: Action of the *SplitFork* enzyme.

As shown in the figure, the system is modified by adding one object, connected bidirectionally to the central one by labelled links. Recognizing forks in the system, that is, pairs of links with non-fundamental adjoints, motion with trace is applied twice to produce the third system. Finally, the missing adjoints are declared fundamental and the labelled links are extracted from the system.

The action of this enzyme produces locally a copy of the original system. In the case of finite bidirectionally connected systems it is shown that propagating through the whole system, the *SplitFork* enzyme produces two isolated, isomorphic systems by finite local operations [30].

2.6 COMPOSITE OBJECTS

Naively, a composite object is an object within a system, that has some internal structure. Since structure in our context is always encoded in form of a system, a **composite object** is an object, which is itself a system. The distinction reflects the scale, or, more generally, the level of abstraction on which the composite object is considered.

From a dynamical point of view, a composite object is a system, that behaves like an object under certain circumstances. This means, that the composite objects delegates some functionality to other objects, while consolidating their behavior in a way, that makes it look like an ordinary object, when considered from the outside.

However, some of the internal structure of a composite object may be visible even on the same level of abstraction. This means, that there are links from the constituent objects to the composite. A collection of arrows, that originate from a constituent and target the composite object, is called an **interface**. The sources of an interface are the **visible** constituents of a composite objects, while the links that belong to an interface are called **ownership relations**. Ownership relations may be classified according to their dynamical behavior as strong or weak ownerships. **Strong ownership** means that if the composite object and its constituents are simultaneously present in the initial system and the dynamics cause the elimination of the composite object, its constituents are eliminated, too. In **weak ownership relations**, the constituents may survive the elimination of the composite.

Composite objects provide a natural reduction of the complexity of a system. The complexity of a system is a measure for the relation of the system's size (for example, the number of objects) and the degree of interconnections of objects in the system. A (dynamical) system is called complex, if its behavior cannot be reproduced by considering subsystems with only a few objects. There are many ways to assign a complexity measure to a system, the choice depending on the interpretation. A simple, universal measure is the *McCabe cyclomatic complexity metric* M [45], which is used frequently in information systems. It simply relates the number of objects n to the number of links l by

$$M := l - n + 2 \quad .$$

In general, the definition of composite objects reduces both, the number of links and the number of objects. However, in most cases the number of links in a composite object is larger then the number of constituents, resulting thereby in a decrease of the McCabe number, the more composite objects arise. In this sense, composite objects provide the main concepts for a **multiscale analysis** of a complex system. Starting with an initial system (finer scale), one defines a composite object by identifying subsystems according to some characteristic properties. The system consisting of the composite objects (coarse scale) then should provide an effective dynamics, which represents only some distinguished properties.

This means, that new enzymes must be defined on the coarser scale.[10] Proceeding on the new scale by further identifications of composite objects, the complexity is further reduced.

Definitions

In the following, a somewhat more precise suggestion for the definition of composite objects is proposed, which is based exclusively on the structure of the underlying category. As argued at the end of the section, it is expected to use more specialized definitions, once additional structure is assumed. However, here the crucial idea is to consider a subcategory as a boundary, if it separates the entities in the containing category into external or internal ones. A composite object then is defined by representing the category in a way that the internal structure is substituted by a single object. In general, it is highly non-trivial to investigate the appropriate conditions for a given scenario, since this preassumes a thorough understanding of the composition process. Here, some remarks concerning the formal definition of composite objects are stated. In the sense suggested here, the definition of a composite object, and thereby the manifestation of structure in the category, depends on the representation category, that means, on the interpretational framework that is used for the definition. This reflects the more physical fact, that what is called a composite object depends on the relational aspects it has to its surrounding, that is, on the arrows in the representation category. Roughly speaking, one may conclude, that structure is in the eye of the beholder.

Boundaries

For a more precise statement of the above considerations, one has to handle the arrows connecting a subcategory $\mathcal{K}_0$ with its surrounding. We denote by $\alpha(f) = X$ and $\omega(f) = Y$ the **source** and the **target** of $f \in \mathrm{Ar}(X, Y)$ respectively. By the **boundary** of $\mathcal{K}_0$ in $\mathcal{K}$ we denote the subcategory $\partial\mathcal{K}_0$ of $\mathcal{K}$ consisting of arrows connecting objects in $\mathcal{K}_0$ to objects outside and of their sources and targets in $\mathcal{K}$. More formally we set

$$\begin{aligned} \mathrm{Ob}(\partial\mathcal{K}_0) &:= \{\alpha(f), \omega(f) \ : \ f \in \mathrm{Ar}[\mathcal{K}_0, *] \setminus \mathrm{Ar}(\mathcal{K}_0)\} \\ \mathrm{Ar}(\partial\mathcal{K}_0) &:= \bigcup_{X \in \mathrm{Ob}(\partial\mathcal{K}_0)} \{\iota_X\} \ \cup \ \Big(\mathrm{Ar}[\mathcal{K}_0, *] \setminus \mathrm{Ar}(\mathcal{K}_0)\Big) \ . \end{aligned}$$

Since due to the definition of the arrows in $\partial\mathcal{K}_0$ this is in a sense a "directed" boundary, we also define the **adjoint boundary** $\partial^*\mathcal{K}_0$ of $\mathcal{K}_0$ in $\mathcal{K}$ to be the subcategory with objects and arrows as follows

$$\begin{aligned} \mathrm{Ob}(\partial^*\mathcal{K}_0) &:= \{\alpha(f), \omega(f) \ : \ f \in \mathrm{Ar}[*, \mathcal{K}_0] \setminus \mathrm{Ar}(\mathcal{K}_0)\} \\ \mathrm{Ar}(\partial^*\mathcal{K}_0) &:= \bigcup_{X \in \mathrm{Ob}(\partial^*\mathcal{K}_0)} \{\iota_X\} \ \cup \ \Big(\mathrm{Ar}[*, \mathcal{K}_0] \setminus \mathrm{Ar}(\mathcal{K}_0)\Big) \ , \end{aligned}$$

[10] Some enzymes remain invariant, that is, they act the same way on all scales. The *SplitFork* of the last chapter is an example.

that is, with arrows having their targets in $\mathcal{K}_0$.
Note that in general $\partial\mathcal{K}_0$ and $\partial^*\mathcal{K}_0$ will *not* consist of the same objects.

For familiar boundary operators in topology it is known, that *boundaries possess no boundary*. Obviously, this holds too for the defined ∂ and ∂^*. To see this formally, consider the objects of the subcategory $\partial\partial\mathcal{K}_0$. An object $X \in \mathrm{Ob}(\partial\partial\mathcal{K}_0)$ of the boundary of $\partial\mathcal{K}_0$ must also belong to the boundary itself, since an arrow to X could be composed to an arrow from an object in $\mathcal{K}_0$ to X. Applying the same argument for ∂^* we get

$$\partial\partial\mathcal{K}_0 = 0 \quad \text{and} \quad \partial^*\partial^*\mathcal{K}_0 = 0$$

and from this, the expected relations for boundary and adjoint boundary operators

$$\begin{aligned} \partial^2 &= 0 \\ \partial^{*2} &= 0 \ . \end{aligned}$$

The dimension of manifolds (or the rank in the case of differential forms) enables a discrete labelling of their associated boundary operators which in turn allows the construction of homological complexes. For subcategories there is no such natural grading and therefore no homology sequences can be built up from the above operators ∂ and ∂^*. Nevertheless the question arises, if there exist subcategories without boundaries which are not boundaries themselves. Disconnected parts of a category have no boundary but must not be boundaries themselves. To examine this question let us analyze the structure boundaries have.
For an arbitrary category $\mathcal{K}$ we define $I_\mathcal{K}$ to be the set of identities and denote its complement in $\mathrm{Ar}(\mathcal{K})$ by

$$A_\mathcal{K} := \mathrm{Ar}(\mathcal{K}) \setminus I_\mathcal{K} \ .$$

Due to their definition each object of a boundary $\partial\mathcal{K}_0$ is either the source or the target of an arrow in $A_{\partial\mathcal{K}_0}$. Defining

$$\begin{aligned} \mathrm{S}_\mathcal{K} &:= \alpha[A_\mathcal{K}] \\ \mathrm{T}_\mathcal{K} &:= \omega[A_\mathcal{K}] \end{aligned}$$

the set $A_{\partial\mathcal{K}_0}$ separates into two disjoint sets and we get

$$\mathrm{S}_{\partial\mathcal{K}_0} \cap \mathrm{T}_{\partial\mathcal{K}_0} = \emptyset \ .$$

We state the following proposition:

Proposition:
Let $\mathcal{C}$ be a subcategory without boundary $\partial\mathcal{C} = 0$. Then $\mathcal{C}$ itself is a boundary, if and only if

$$S_\mathcal{C} \cap T_\mathcal{C} = \emptyset \ . \tag{2.6.1}$$

Proof:
Since for boundaries the condition is fulfilled we construct a subcategory $\mathcal{K}_0$ with boundary $\mathcal{C}$. From (2.6.1) we define $\mathcal{K}_0$ by $\mathrm{Ob}(\mathcal{K}_0) := \mathrm{S}_\mathcal{C}$. Then

$$\begin{aligned} \mathrm{Ob}(\partial\mathcal{K}_0) &= \mathrm{S}_\mathcal{C} \cup \mathrm{T}_\mathcal{C} \\ \mathrm{Ar}(\partial\mathcal{K}_0) &= \mathrm{Ar}(\mathcal{C}) \end{aligned}$$

and we get the desired relation $\mathcal{C} = \partial\mathcal{K}_0$.

□

The same result holds also in the case of adjoint boundaries ($\partial^*\mathcal{C} = 0$) where the condition for the existence of a subcategory with adjoint boundary $\mathcal{C}$ is exactly the same as (2.6.1).

Composite objects

We proceed to a definition for composite objects. Note, that the definition below relies on the category structure and does not make use of the system structure. For this, let $\mathcal{K}_0$ be a subcategory of $\mathcal{K}$.

> We call $\mathcal{K}_0$ a **composite object** in $\mathcal{K}'$, if there exists a representation ρ onto a category $\mathcal{K}'$ and a specific object $X'_0 \in \mathrm{Ob}(\mathcal{K}')$, having the following property
>
> $$\rho(X) = X'_0 \;\Leftrightarrow\; X \in \mathrm{Ob}(\mathcal{K}_0) \quad . \tag{2.6.2}$$

The definition is illustrated schematically in fig. 2.2.

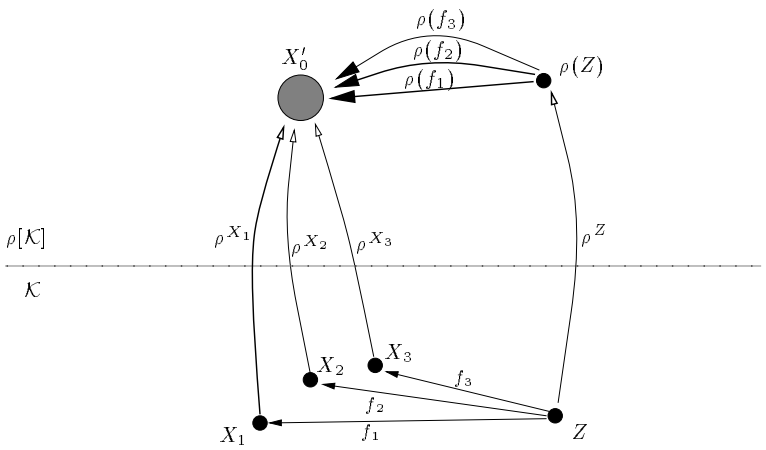


Figure 2.2: Representation and composite object.

From the definition, the properties

$$(i) \quad \rho\left(\mathrm{Ar}\left(\mathcal{K}_0\right)\right) = \mathrm{G}_{X_0'} \tag{2.6.3}$$

$$(ii) \quad \rho\Big(\mathrm{Ar}(\partial\mathcal{K}_0)\Big) = \mathrm{Ar}(\partial X_0')$$

$$\rho\Big(\mathrm{Ar}(\partial^*\mathcal{K}_0)\Big) = \mathrm{Ar}(\partial^* X_0') \ , \tag{2.6.4}$$

follow immediately. These properties ensure, that all internal arrows become elements of the local gauge semi group of the composite object and that all arrows relating internal objects to external ones become arrows from or to the composite object.

A **scale** is a representation category, that defines composite objects for all objects of the represented category, that is, every object becomes a constituent in a composite object.

A natural way for composite objects to occur is in the case of unfrustrated subcategories. For this we define a category to be **connected**, if there are no isolated parts, that is, if there exists no subcategory possessing no arrow from or to objects not belonging to it.
Further on we define a category to be **bidirectionally connected**, if it is connected and there exists for each object at least one arrow going to as well as one starting from this object. This means

$$\mathrm{Ar}(X,Y) \neq \emptyset \ , \ \forall X, Y \in \mathrm{Ob}(\mathcal{K})$$

for a bidirectionally connected category.

With this definitions we get the following proposition:

Proposition:
Every bidirectionally connected, unfrustrated subcategory $\mathcal{K}_0$ is a composite object.

Proof:
Let $X_0 \in \mathrm{Ob}(\mathcal{K}_0)$ be an object in $\mathcal{K}_0$. We construct a representation ρ onto a category $\mathcal{K}'$ consisting of objects

$$\mathrm{Ob}(\mathcal{K}') := \{X_0\} \cup \Big(\mathrm{Ob}(\mathcal{K}) \setminus \mathrm{Ob}(\mathcal{K}_0)\Big) \ .$$

The representation is defined to act as the identity on objects $X \notin \mathrm{Ob}(\mathcal{K}_0)$ and as

$$\rho(X) = X_0 \ \Leftrightarrow \ X \in \mathrm{Ob}(\mathcal{K}_0) \ .$$

Arrows $f \in \mathrm{Ar}(\mathcal{K}_0)$ are mapped on the identity $\iota_{X_0} \in \mathrm{Ar}(\mathcal{K}')$ of the composite object, those between objects $X, Y \notin \mathrm{Ob}(\mathcal{K}_0)$ are leaved unchanged.

Arrows of the boundary and the adjoint boundary of $\mathcal{K}_0$ are represented as follows: Since $\mathcal{K}_0$ is bidirectionally connected and unfrustrated there exists unique arrows

$$C_{YX} : X \to Y$$

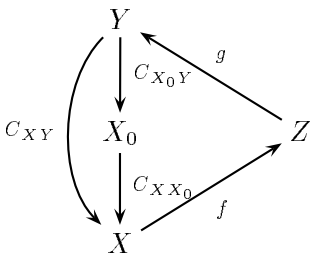
for all $X, Y \in \text{Ob}(\mathcal{K}_0)$.
Let $f \in \text{Ar}(\partial\mathcal{K}_0)$ and $g \in \text{Ar}(\partial^*\mathcal{K}_0)$, we set

$$\begin{aligned} \rho(f) &:= f \circ C_{\alpha(f)X_0} \in \text{Ar}(\partial X_0) \subset \text{Ar}(\mathcal{K}') \\ \rho(g) &:= C_{X_0\omega(g)} \circ g \in \text{Ar}(\partial^* X_0) \subset \text{Ar}(\mathcal{K}') \end{aligned}$$

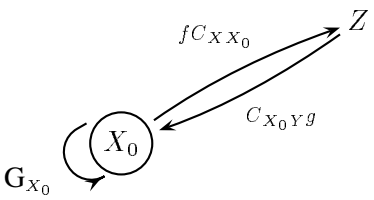
and for the composition of arrows

$$\begin{aligned} \rho(f) \circ \rho(g) &= f \circ C_{\alpha(f)X_0} \circ C_{X_0\omega(g)} \circ g \\ &= f \circ C_{\alpha(f)\omega(g)} \circ g \ . \end{aligned}$$

The construction is shown graphically in the following diagram:



This yields the following diagram in the category $\mathcal{K}'$:



Further remarks

To illustrate the handling of composite objects in more specific situations, some remarks are stated:

- If the category $\mathcal{K}$ possesses a system structure $\mathcal{S}$, the representation ρ defining a composite object is expected to preserve the system structure at least outside the composite object. The composite object should originate from a subsystem $\mathcal{S}_0$ in $\mathcal{S}$.

- One may further require the representation ρ to be an intertwiner. Then, the relations

$$X_i \stackrel{\rho^{X_i}}{\to} X_0 \quad ,$$

for objects $X_i \in \mathcal{S}_0$ are arrows from the constituents to the composite object, defining an interface, which makes all constituents visible and fulfills

$$\rho^{X_i} \circ f_i = \rho(f_i) \circ \rho^Z \quad ,$$

for the situation illustrated in fig. 2.2.

Consider for example the setting of the last proposition. Defining the arrow

$$\rho^Y \ := \ C_{X_0 Y} \quad ,$$

for all constituent objects $Y \in \mathrm{Ob}(\mathcal{S}_0)$, turns the representation ρ into an intertwiner. This is easily seen, by the definition of the representation for arrows in the boundary, or adjoint boundary of the subcategory $\mathcal{K}_0$. Since $\mathcal{K}_0$ is bidirectionally connected and unfrustrated, this is a unique assignment of an ownership relation for each constituent.

Result:
Every bidirectionally connected, unfrustrated subcategory is a composite object with an interface, declaring all of its constituents as visible objects.

- Consider a representation ρ of a system $\mathcal{S}$, that defines a single composite object $\rho[\mathcal{S}_0]$. The representation is assumed to be an isomorphism when restricted to the external subsystem $\mathcal{S}_0'$. If the restricted isomorphism is an intertwiner, one results with links connecting external objects Z and their images $\rho(Z)$. These links possess adjoints, that are defined by the inverse isomorphism, the links are unitary. Then, the layered system $\mathcal{S} \cup \rho[\mathcal{S}]$ with inter-layer-links the unitary ρ^Z for external Z, contains composite objects, since every external object Z may be identified with its image. This is possible, since the Z and $\rho(Z)$ connected by the unitary links, build a bidirectionally connected, unfrustrated subsystem, so the proposition applies. In this case, constituents and the composite objects are considered to lie in the same system.

Moreover, in the same setting, if the system $\rho[\mathcal{S}]$ is unfrustrated, the composite object and its interface build a coproduct of the constituents in the category $\mathcal{K} \cup \rho[\mathcal{K}]$. The links b are the ones in the unfrustrated system $\rho[\mathcal{S}]$, the unique arrow h to an arbitrary object C is given by

$$h := b(X_0 \to \rho(Y)) \quad ,$$

if the object C is in $\rho[\mathcal{S}]$, and by

$$h := (\rho^Y)^* \circ b(X_0 \to \rho(Y)) \quad ,$$

if Y is in $\mathcal{S}$. This is due to the fact, that all inter-layer-links are paths, containing at least one unitary ρ^Y or its adjoint.

Finally, every communication network contains composite objects. The idea is to define the 'black-boxes', containing the maps X and their sources and targets, as interconnected objects. For this, define the category $\mathcal{K}'$ consisting of objects B_X and arrows f' defined as follows

$$B_X = (X, A_X, \Omega_X) \quad ,$$

with invertible functions

$$X : A_X \to \Omega_X \quad .$$

The arrows $\mathrm{Ar}(B_X, B_Y)$ are functions $f : \Omega_X \to A_Y$, which are denoted by f' when considered as arrows. The composition of arrows f', g' at B_X is given by

$$f' \circ'_{B_X} g' := f \circ_X X \circ_X g \quad ,$$

where the right hand side is considered for functions f and the left hand side for the corresponding arrows f'. The identities are defined by

$$\imath_{B_X} := \left(X^{-1}\right)' \quad .$$

Now, suppose a communication network $\mathcal{C} = (X, A_X, \Omega_X, f)$ is given. Considering $\mathcal{C}$ as a system with X and f as fundamental arrows, this generates a category by the ordinary composition of functions. The functor $\rho : \mathcal{C} \to \mathcal{K}'$ is then defined by

$$\begin{aligned} \rho(A_X) &:= \rho(\Omega_X) = B_X \\ \rho(X) &:= \imath_{B_X} \\ \rho(f) &:= f' \quad , \end{aligned}$$

and is extended homomorphically to $\mathrm{Ar}(\mathcal{C})$. Restricting ρ to its image $\rho[\mathcal{C}]$, one gets composite objects B_X, representing the input-output structure of the communication network as a 'black-box'.

2.7 SUMMARY

In this chapter the basic concepts for the analytical treatment of systems have been explained. Relying on mathematical categories and corresponding functors, the notions of representations and gauge transformations are explained. Systems were introduced as categories with a locality principle. The typical dynamics of systems were explained. Finally, to enable multiscale analysis, a definition for composite objects was suggested and some properties were discussed.

Chapter 3

ToCS-based Formal Modeling

In this thesis, systems are thought to be either, physical systems, or information systems. Although in modeling a physical system it may be sufficient to choose a single mathematical framework for specifying it, in information systems the choice of appropriate models is a non-trivial task. Moreover, the analysis of an information system as a network may require the simultaneous usage of various methods. To describe simple systems, automata-based techniques, like P/T-nets may be sufficient, but for generic complex systems the rich structure requires more advanced modeling approaches.

Advanced modeling settings like the UML are discussed in later chapters. Here some frequently used approaches are described and related to the ToCS framework. The ToCS-based formulation of various techniques, illustrates the universal character of its concepts.

3.1 Formal Models

By modeling, one tries to achieve a simplification of the structure and behavior of a 'real' system. In general, every non-trivial system is best approached through a (small) set of nearly independent models. Elements of the original system with brand effect should be included in appropriate form in the models, whereas elements, that are not relevant for the specific level of abstraction, should be omitted. Consequently, to describe different aspects of a system, one uses more than one model simultaneously. For this, the consistency of the different models must be assured. The simultaneous usage becomes more and more important, the larger the complexity of the modeled system is. The internal consistency, as well as the consistency between the different models is guaranteed best by using formal modeling techniques. In particular, considering information systems, the usage of formal models increases the safety of the system not only in its present status but also of a future system, arising from the present one by well-defined modification and extension mechanisms. Yet, since the support of different aspects varies significantly in different modeling approaches, it is non-trivial to decide, which modeling approach should be used. However, even after

specifying an approach, it is useful to enable the consideration of a model at different scales, since the complexity in a model may still be large.

For the purposes that are considered here, the following definition of views and models is suggested.

> **Definition (View, Model):**
> Given a system $\mathcal{S}$. A **view** of $\mathcal{S}$ is a representation ρ of a subsystem $\mathcal{S}' \subset \mathcal{S}$ of the system. A **model** $\mathcal{M}(\mathcal{S})$ is a collection of views ρ_i with corresponding subsystems $\mathcal{S}'_i$, that describe the system, that is,
>
> $$\mathcal{S} = \bigcup_i \mathcal{S}'_i \quad .$$

Note, that the union need not to be disjoint. Since they describe the same system, it is reasonable to expect the consistency of overlapping views to be assured by the existence of corresponding intertwiners. A system may be considered as a model of itself. Further, every representation is considered as a model of the represented system. The crucial point in the definition is, that it may be impossible to represent the whole system in a single view.

Examples

(i) The decomposition of a system into consistent and possibly overlapping pieces is used in many modeling approaches. In general, separating parts of the system into units, which encapsulate some structure, while interacting with their environment in specific way, is called modularization. In this sense, the usage of views supports the conceptual modularization of a system. This is used for example in data base modeling [46], where a conceptual model is composed from several external models, in the viewpoint concept in general system engineering [47].

(ii) In a recent paper, a view concept was proposed for graph transformation systems [48]. Graph transformations are used in the system modeling area in various contexts [49], for example as entity-relationships graphs, or in the object-orientation paradigm, to specify the relations between different elements of the model. However, graphs arise in this frameworks as pairs of graphs (Γ_S, Γ_I), where Γ_S is called the scheme graph, and Γ_I the instance graph. The scheme graph represents the conceptual structure of the system, lying thereby on a higher level of abstraction than the instance graph. One may think for example of an unfrustrated system consisting of sets and functions as the scheme graph. The instance graph is then a graph, that consists of elements of the sets as its objects, related by the functional relation to other elements. The instance graph must be compatible with the scheme graph. For our purposes, the instance graph is thought to allow a functor to the scheme graph, where composite objects may occur (previous to the functorial map).

Given two graphs Γ, Γ', a *rule-based graph transformation* is a transformation of graphs

$$\Gamma \xrightarrow{r} \Gamma' \quad ,$$

which acts on Γ by removing a subgraph and adding another graph to the result. More formally, the rule-based graph transformation is a tuple

$$(r, L, R, m, m^*) \quad ,$$

where r is the name of the transformation, L, R and $L \cap R$ are graphs, called *left-*, *right-hand side*, and *interface*, respectively. The m, m^* are graph homomorphisms

$$\begin{aligned} m &: L \to \Gamma \\ m^* &: R \to \Gamma' \quad , \end{aligned}$$

called *match* and *comatch*, respectively. The transformation r than acts on Γ as follows

$$\Gamma \xrightarrow{r} \Gamma' := \Big(\Gamma \setminus m(L \backslash R)\Big) \cup m^*(R \backslash L) \quad .$$

The action assures, that unaffected parts of the graphs do not change names. A *graph transformation system* G is a tuple (Γ_S, N, ρ) consisting of a scheme graph Γ_S (which may be thought together with a collection of instance graphs), a set of rule-names N and a mapping ρ assigning to each rule-name a rule-based graph transformation. In a graph transformation system, the left- and right-hand side are instance graphs of the scheme graph. A view of a graph transformation system G is defined by a pair (v, G_0), where G_0 is a graph transformation system called the *view*, and v is called the *view relation*, which represents the following sequence

$$v = (G_0 \overset{ren}{\leftrightarrow} G_0^1 \hookrightarrow G) \quad ,$$

the first bidirectional arrow represents a simple renaming of G_0, while the second symbolizes, that G extends G_0^1, that is, the corresponding scheme graphs fulfil a subgraph relation, while the name-sets fulfil a subset relation. For the rules, the extension means,

$$r \in N_0^1 \subset N \Rightarrow \rho_0^1(r) \subset \rho(r) \quad ,$$

where the relation on the right hand side denotes, that

$$\begin{aligned} L_0^1(r) &\subset L(r) \\ R_0^1(r) &\subset R(r) \quad , \end{aligned}$$

for the corresponding graphs. A view here does not only provide a projection of the original structure of the scheme graph Γ_S of G, but also of the dynamics of the corresponding rules.[1] Since graphs may be considered as systems, this extends the definition of views given above.

[1] In [48], an automatic integration of various views is demonstrated, too.

3.2 MODELING BEHAVIOR

In this section, some techniques, to model the behavior of systems are discussed.

3.2.1 STATE-TRANSITION SYSTEMS

Automata-based techniques are frequently used in the modeling of systems. The idea is, to describe a system by a state space, wherein the dynamics of the system is given by an evolution path. The path describes the changes of the system's state as a reaction on a given stimulus, while transmitting a specific response. The system is then specified by a so-called state-transition system, containing the states that may occur during the 'lifetime' of the system, related with each other by corresponding state-changes. In this sense, a **state transition system** $\mathcal{S}$ consists of a set Z of **states** as its objects and fundamental arrows

$$\text{Fund}(\mathcal{S}) \subset Z \times Z \quad ,$$

called (state) **transitions**, together with the formal identity-loops at each object. Of course, every relation in a set defines a trivial state-transition system. However, states in information systems usually arise by variable binding. For this, a collection of sets A_i is given (domains), together with a set R of relations between the domains

$$R \subset \{A_i\} \times \{A_j\} \quad ,$$

called the *structure relations*. The states $\text{Ob}(\mathcal{S})$ are then given as a subset of possible tuples

$$\text{Ob}(\mathcal{S}) \subset \ \times_i A_i \quad ,$$

where the structure relations R are used to constrain the product of the domains A_i. A **state-transition system** is a system structure on $\text{Ob}(\mathcal{S})$, that is, the specification of a relation

$$\mathcal{T} \subset \text{Ob}(\mathcal{S}) \times \text{Ob}(\mathcal{S}) \quad ,$$

defining the set of direct state-changes. Usually, the transitions in $\mathcal{T}$ carry additional parameters, like transition conditions, stimuli, and so on.

Example

Consider a $m \times n$ lattice-graph Γ with $\mathbb{Z}_2$ assigned to each vertex. The structure relations are the directed edges of the lattice, to which the constraints $[s]$ or $[a]$ may be attached. They describe, if neighbors are constrained to have equal or opposite signs, respectively. Then, the states are given by $\mathbb{Z}_2$-valued mn-tuples, fulfilling the constraints, while the transitions represent valid changes of values in the vertices.

State-transition systems are discussed in more detail in chapter 5, where they appear as object state machines in the context of the Unified Modeling Language (UML) approach for complex systems modeling. There, they are represented by statecharts and respect the object-orientation paradigm. They are used for modeling both, the internal state changes of objects in an information system, and of processes occurring in the system.

3.2.2 PETRI NETS

Concurrent and distributed systems, consisting of internal actions and connections to their surroundings lie in the focus of various modeling approaches. One of the first approaches to handle interacting processes in simple, universal and graphical form have been Petri nets [50, 51].

Usually one distinguishes between low-level and high-level Petri nets. Low-level Petri nets consist of elementary constituents (tokens), whereas high-level version are based on more complex constituents, for example algebraic, or otherwise specified entities [52]. The simplicity of the low-level versions cause a lack of these approaches in describing and structuring more complex data types. Further, there is no generic way to compose and refine low-level nets non-trivially. These issues require the usage of high-level versions [53]. For this, the simple marking by tokens is substituted by algebras, the transitions then correspond to algebra transformations. Moreover, Petri nets possess a natural monoid structure [54]. Their algebraic structure is based on their form as directed graphs with algebraic operations defined on them. This is used to define representations of P/T-nets based on mathematical categories and are thereby analyzed by abstract algebra methods. Both, the usage of higher-level Petri nets, and the formulation of Petri nets in categorical fashion, serves the bypassing of the mentioned inabilities, common to almost all low-level modeling technique.

This section provides an alternative, ToCS-based formulation of P/T-nets. It evades the mentioned inabilities. It describes the basic properties of specific low-level Petri nets, called P/T-nets, while relating their structure and dynamics to the general ToCS framework.

P/T-nets

Place/Transition nets (P/T-nets) are used to describe the flow of material entities, or information along certain channels in a system. They model concurrent processes by elementary rules with simple graphical notation. Due to their universality and simplicity when considered as a system specification technique, many computer-based tools in system engineering support P/T-nets. P/T-nets are composed of two basic entities, a collection of places and a collection of transitions. The places and transitions in a P/T-net model the possible changes of the states of a system, as shown below. Places and transitions may be related to each other. In contrast, places cannot be directly related to any other place and transitions cannot be directly related to any other transition. More formally, the structure of a P/T-net is expressed by the following definition:

Definition (P/T-net):
A P/T-net is a tuple $(\mathcal{P}, \mathcal{T}, \Phi)$, where $\mathcal{P}$ is a set called the **places** and $\mathcal{T}$ is a set called the **transitions** of the P/T-net. Places and transitions are disjunct sets,

$$\mathcal{P} \cap \mathcal{T} = \emptyset \quad .$$

The binary relation F

$$\Phi \subset (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$$

is called the **flow relation**.

To introduce the dynamics of P/T-nets, the notion of **pre-condition** $\bullet X$ of a place or transition, and the **post-condition** $X \bullet$ are defined. For every place or transition $X \in \mathcal{P} \cup \mathcal{T}$ in the P/T-net, they are defined as follows

$$\begin{aligned} \bullet X &:= \{Z \ : \ (Z, X) \in \Phi\} \\ X \bullet &:= \{Y \ : \ (X, Y) \in \Phi\} \quad . \end{aligned}$$

Pre-conditions and post-conditions are important to specify, which transitions are allowed to take part in a dynamical evolution.

Remark:
A P/T-net containing one and only one place in the pre- and post-condition of any transition in the P/T-net is a state-transition system.

In essence, a P/T-net is a bipartite, directed graph, separating into the regions $\mathcal{P}$ and $\mathcal{T}$ that are connected by Φ. Therefore, the data of a P/T-net may be considered as a trivial system. The objects are the union of places and transitions, while the links are given by the elements of the flow relation and formal identities at each object. The composition is the formal path-building of links. Pre- and post conditions then correspond to the neighborhood of the considered object, distinguishing objects pointing towards the object and objects, the current object points to.

The dynamics of a P/T-net is given by a sequence of markings, which evolve from each other by rules encoded in the structure of the P/T-net. A marking assigns a number of tokens to each place in the P/T-net. The distribution of tokens evolves dynamically by the firing of some transitions. A transition may only fire, if it is enabled by the current marking, that is, if each place in its pre-condition contains at least one token. The firing of a transition moves the tokens through the net, by removing them from its pre-condition and adding them to its post-condition. The moving of tokens from different regions of the P/T-net models the dynamics of processes defined by the structure of the net. More formally, markings and firing is defined as follows

Definition (Marking, Firing):
A map $m : \mathcal{P} \to \mathbb{N}$ is called a **marking** of the P/T-net. A transition T is called m**-enabled**, if and only if

$$m(P) > 0 \quad , \quad \forall P \in \bullet T \quad .$$

A m-enabled transition T may cause a **follower marking** $m' = T(m)$, which is given by the following conditions

$$\begin{aligned} m'(P) &:= m(P) - 1 \quad \Leftrightarrow \quad P \in {\bullet}T \backslash T{\bullet} \\ m'(P) &:= m(P) + 1 \quad \Leftrightarrow \quad P \in T{\bullet} \setminus {\bullet}T \\ m'(P) &:= m(P) \quad \text{otherwise} \quad . \end{aligned}$$

The transition T is then said to be **firing**, denoted by $m \xrightarrow{T} m'$.

The dynamics of a P/T-net is given by an initial marking m and sequence of transitions T_i, where T_i is a $T_{i-1}...T_1(m)$-enabled transition. However, from the definition it follows immediately, that the number of tokens

$$\sum_{P \in \mathcal{P}} m(P)$$

is not invariant under the dynamics. Although the structure of a P/T-net may be highly complex, its dynamics may be explained by some elementary firings. To illustrate the dynamics of a P/T-net, the firing at a fork and at a join is shown in fig. 3.1.

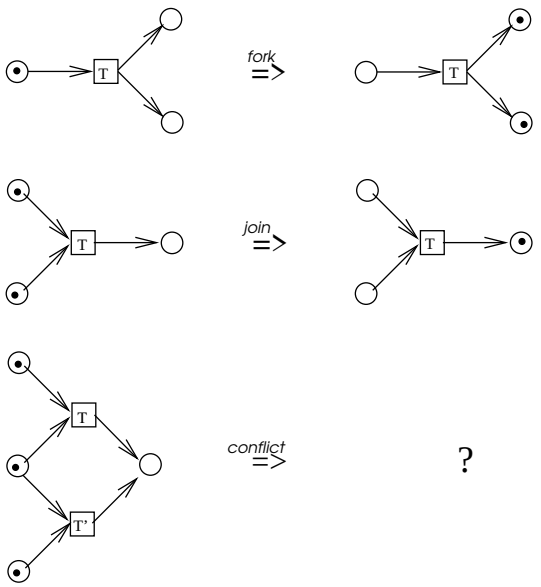


Figure 3.1: The elementary firings in a P/T-net. The first firing represents a fork transition, the second a join. The third situation is called a conflict scenario, since one has to decide, which transition should fire.

As mentioned, the dynamics in a P/T-net are given by composing the elementary moves of fig. 3.1 into more complex structures. The tokens move from the pre-conditions to the

post-conditions of the enabled transitions. The third scenario shows a conflict. In general, simultaneously enabled transitions may fire parallel, but in conflict this is to be handled with care. In a conflict scenario, the firing of one transition results in a follower marking, that does not enable the other transition. The parallel firing of transitions should be used only, when no conflict is present. For this, suppose n transitions T_i are given. The transitions are called **independently enabled** by a marking m, if for all subsets of the transitions $\{T_i\}$ the following condition hold

$$m(P) > k \quad , \forall P \in \bigcap_{a=1}^{k} \bullet T_{i_a} \quad ,$$

where $\{T_{i_a}\}$ is a subset containing k transitions. The conditions must hold for all $k \leq n$, that is, there are 2^n conditions to be fulfilled. Transitions that are not independently enabled, are said to be in **conflict**. In this case, an enabled transition selection function decides, what the firing order should be. Concurrent systems are modeled by using parallel firing of independently enabled transitions in P/T-nets. For simplicity, here it is assumed, that the firings in a P/T-net arise only serial, and their order is given by an explicit sequence. The firing of a transition may result in a marking that does not enable any further transition to fire. This is called a **deadlock** situation.

The definition of P/T-nets given here, may be generalized by weighted flow relations. A **weight** is a function $w : \Phi \to \mathbb{N}$, that produces the follower marking of an enabled transition T as follows

$$\begin{aligned}
m'(P) &:= m(P) - w(P,T) \quad \Leftrightarrow \quad P \in \bullet T \backslash T \bullet \\
m'(P) &:= m(P) + w(T,P) \quad \Leftrightarrow \quad P \in T \bullet \setminus \bullet T \\
m'(P) &:= m(P) - w(P,T) + w(T,P) \quad \Leftrightarrow \quad P \in \bullet T \cap T \bullet \\
m'(P) &:= m(P) \quad \text{otherwise} \quad .
\end{aligned}$$

This generalization does not cause any conceptual changes and is easy to take into consideration. Therefore, the weights are restricted to $w = 1$ here.

State system of a P/T-net

In general, starting with an initial marking m, every follower marking m' that results from the firing of a m-enabled transition T is called **directly reachable**, denoted by $m' = T(m)$. The firing of a m'-enabled transition T' yields the marking $m'' = T'(m')$, which is called **reachable** from m. In this sense, the **reachable set** $< m >$ of a marking is defined as the reflexive, transitive closure of the 'direct reachable' relation, that is $< m >$ is defined by the condition $m \in < m >$ and for all $m_1 \stackrel{T}{\to} m_2$

$$m_1 \in < m > \quad \Rightarrow \quad m_2 \in < m > \quad .$$

Changing the terminology, a marking m is called the **state** of the P/T-net. The union

$$\Sigma := \bigcup_{m:\text{marking}} < m >$$

then is the **state space** of the P/T-net. For a P/T-net consisting of k places, the state space is discrete, $\Sigma = \mathbb{N}^k$. The state space carries a natural system structure. A point m in the state space Σ possesses a fundamental arrow to another point m', if and only if, there exists at least one m-enabled transition T, whose follower marking is m'. The composition of arrows in this system is the ordinary formal path-building. This system is called the **state system** of the P/T-net. The state system need not to be connected. Rather, its isolated parts correspond to semantically distinguished scenarios modeled by the P/T-net. Nevertheless, the reachable set $< m >$ of an initial state corresponds to the connected subsystem $\Sigma_m \subset \Sigma$ in the state system Σ called the **reachable system** of m. The dynamics of the P/T-net, originating from the firing of transitions corresponds to a path in the reachable system of the marking, that enables the transitions initially. For example, periodic dynamics corresponds to loops in Σ, while deadlock situations are represented by objects in the system with no departing arrows. However, restricting the capacity of the places to be finite, the state system Σ becomes a finite subset of $\mathbb{N}^k$ and may be considered as a state-transition system.

To model a specific scenario, a P/T-net is equipped with an initial state defined by a marking m. The P/T-net then specifies by its enabled transitions the evolution of the state of the net.

Definition:
A **marked P/T-net** is a P/T-net together with a state

$$m_0 : \mathcal{P} \to \mathbb{N} \quad ,$$

which is called its **initial state**. The **dynamics** of a marked P/T-net, caused by the firing of transition-sequences, is a path in the reachable system Σ_{m_0} of the initial state m_0, which source is m_0.

The reachable system Σ_{m_0} of the initial state of a marked P/T-net is sometimes also called its **sequential reachability system** in the state system Σ.

Example

A simple P/T-net is given in fig. 3.2. It consists of the places $P1, P2, P3, P4, P5$ and the transitions $T1, T2, T3, T4$. The flow relation is given by the arrows in the figure. The left hand side of the net represents some process, the right hand side another one. The processes are active, if tokens are located at places of the processes. The middle place $P3$ enables the interaction of the processes.
A marking $m : \mathcal{P} \to \mathbb{N}$ of the P/T-net of fig. 3.2 may be given by a 5-tuple of integers. Therefore the state space Σ is given by $\mathbb{N}^5$. For the following states

$$\begin{aligned} m_0 &= (0,1,0,0,1) \\ m_1 &= (1,0,1,0,1) \end{aligned}$$

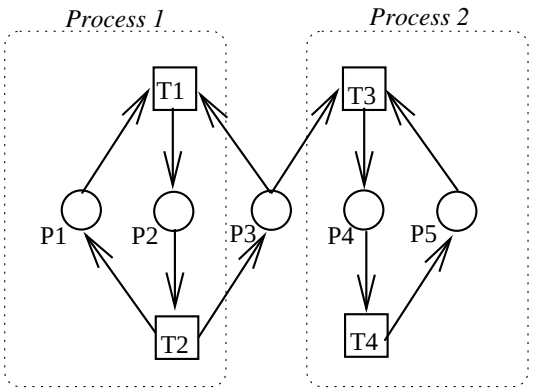


Figure 3.2: A simple P/T-net. The places $P1$ and $P2$ belong to some process, the places $P4$ and $P5$ to some other process. The place $P3$ represents the exchange of information between both processes.

$$\begin{aligned} m_2 &= (1,0,0,1,0) \\ m_3 &= (1,0,0,0,1) \\ n_0 &= (0,1,1,0,1) \\ n_1 &= (1,0,2,0,1) \\ n_2 &= (1,0,1,1,0) \\ n_3 &= (0,1,0,1,0) \end{aligned}$$

the reachable sets $< m_0 >$ and $< n_0 >$ are given by

$$\begin{aligned} < m_0 > &= \{m_0, m_1, m_2, m_3\} \\ < n_0 > &= \{n_0, n_1, n_2, n_3, m_0, m_1, m_2, m_3\} \end{aligned}$$

therefore, the sequential reachability system Σ_{m_0} of the initial marking m_0 is a subsystem of the system Σ_{n_0}. Both systems are shown in fig. 3.3.

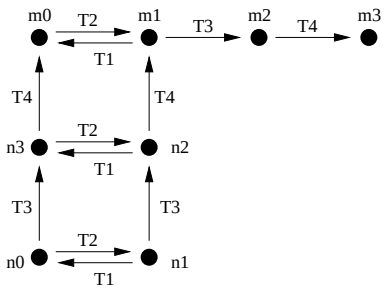


Figure 3.3: The sequential reachability system of the initial states m_0 and n_0. The system Σ_{n_0} contains Σ_{m_0} as a subsystem.

As mentioned, the dynamics of the P/T-net is given by a sequence of transitions. For the

initial marking n_0 the dynamics of the net is given by all paths in the system in fig. 3.3.

ToCS-based formulation of P/T-nets

The data of a P/T-net may be used to define a system that represents the same structure and has the same dynamics. Considering places and transitions as objects in separated layers of a layered system, the links are given from the flow relation of the P/T-net and the formal identities at each object. This is stated more precisely in the following definition.

Definition (P/T-system):
A **P/T-system** is a two-layered system $\mathcal{S} = (\mathcal{P}, \mathcal{T})$, whose layers $\mathcal{P}$ and $\mathcal{T}$ consist of isolated objects. The links of the system are given by a subset

$$\text{Fund}(\mathcal{S}) \subset (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$$

and the formal identities at each object.

Note, that the two layers of a P/T-system cannot be integrated into a single layer by defining a composite arrow from one place to another via a mediating transition. This would require the definition of multiple relationships between more than two objects, which in essence would reconstruct the transitions as composite objects.

From the definition above, it is clear, that a P/T-net corresponds to a P/T-system and vice versa. Given one of them, the other may be constructed from the given data in straightforward manner. The P/T-system and the corresponding P/T-net are said to be associated to each other.

In a P/T-system there is at most one link between a specific transition and a specific place. This simple remark will be used later on, when the dynamics of marked P/T-systems are considered. Like P/T-nets, the dynamics of a P/T-system require the system to be equipped with a marking. In the ToCS-based formulation a marking is given by an additional layer $\mathcal{M}$, consisting of one single object. This object is connected to the places-layer $\mathcal{P}$ by unidirectional links. In general, the links from $\mathcal{M}$ to $\mathcal{P}$ will be frustrated, meaning that more than one link originating from $\mathcal{M}$ may terminate at the same place. The number of links pointing to a place-object, corresponds to the marking of the place in the associated P/T-net. A marked P/T-system is a P/T-system together with an initial marking.

Definition (Marked P/T-system):
A **marked P/T-system** is a three-layered system $\mathcal{S} = (\mathcal{P}, \mathcal{T}, \mathcal{M})$, containing two layers $(\mathcal{P}, \mathcal{T})$, that build a P/T-system. The layer $\mathcal{M}$ consist of one and

only one object M, which is the source of a set of fundamental arrows targeting some places in the places-layer of the contained P/T-system. In general, Fund$(M, \mathcal{P})$ may contain more than ane link to a specific place.

From a marked P/T-system, a marked P/T-net is easily reconstructed. The P/T-net is the one associated to the contained P/T-system, while the **associated marking** m of the system is given by

$$m(P) := |\text{Fund}(M, P)| \quad ,$$

that is, by the number of arrows from M targeting the specific place P.

The notions of pre- and post-condition may be applied to P/T-systems, too. In addition, by using the associated marking, the notions of *states* (=markings), of *reachable sets* and of the *state system* $\Sigma[\mathcal{S}]$ may be used for P/T-systems with more or less no modifications. However, for the dynamics of a marked P/T-system, the notion of states is slightly extended.

Like P/T-nets, once an initial state is determined, a P/T-system may evolve from one state to another by the firing of transitions. To enable the formulation of transition-firing in form of an enzymatic action, the notion of generalized states is introduced. A **generalized state** Ψ of a marked P/T-system $\mathcal{S}$ is a subset of links of the system

$$\Psi(\mathcal{S}) := \text{Fund}(M, *) \cup \text{Fund}(*, M) \quad ,$$

where the subset Fund$(M, \mathcal{T}) \subset \Psi(\mathcal{S})$ is constrained to be unfrustrated. Although for a proper P/T-system the sets Fund$(M, \mathcal{T})$ and Fund$(*, M)$ are always empty sets, violating situations may arise for transient systems, when the dynamics of a marked P/T-system is given by sequences of elementary enzymatic actions. However, the final state, when the action sequence is performed, must be a proper state of the P/T-system. In this sense, a (proper) **state** of a P/T-system is a generalized state $\Psi(\mathcal{S})$, fulfilling

$$\Psi(\mathcal{S}) = \text{Fund}(M, \mathcal{P}) \quad ,$$

otherwise, $\Psi(\mathcal{S})$ is called a **pseudostate** of the P/T-system. The dynamics of a marked P/T-system should then provide a well-defined path in the state system $\Sigma[\mathcal{S}]$.

The dynamics of marked P/T-system is performed by the firing of transitions. A generic ToCS-based formulation of P/T-nets requires not only the definition of the net as a system. Moreover, the firing of transitions must be defined as a sequence of enzymatic actions. In a P/T-system, a transition T may fire only if it is **enabled** by the (proper) state $\Psi(\mathcal{S})$ of the system, that is,

$$|\text{Fund}(M, P)| > 0 \quad , \quad \forall P \in \text{Fund}(*, T) \subset \Psi(\mathcal{S}) \quad .$$

Before defining the firing of transition in an arbitrary P/T-system, the mechanism is illustrated in a simple example, shown in fig. (3.4).

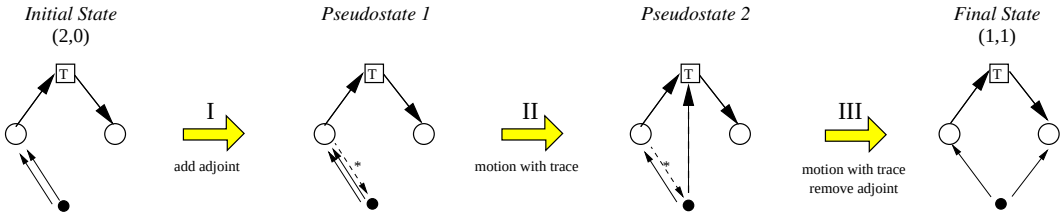


Figure 3.4: The dynamics of a simple P/T-system consisting of two places, with one mediating transition. The elementary actions are defined locally and lead to an evolution of the marking of the system that corresponds to the firing of the enabled transition T.

The system consists of two places $P1, P2$ and one mediating transition T. The firing of the transition is performed in three elementary steps [III.II.I], where the notation should be read from the right. After the action of I and II, the system is in a transient pseudostate. It returns to a proper state after completing of the action III. The actions are defined as follows:

I Initial State $\rightarrow$ Pseudostate 1
Since T is enabled, the place $P1$ in its pre-condition is the target of at least one link from M. At the place $P1$ in the pre-condition, one and only one adjoint link, pointing towards M is added to the system. After completing, the system is left in the pseudostate 1.

II Pseudostate 1 $\rightarrow$ Pseudostate 2
Motion with trace is performed at M. Since Fund(M, T) is unfrustrated by the definition of pseudostates, one new link from M to T is declared fundamental. The system is left in the pseudostate 2.

III Pseudostate 2 $\rightarrow$ Final State
Motion with trace is performed at the place $P1$, removing the adjoint. Motion with trace at M leaves the system in the final state with one link from M to the place $P2$, representing the movement of the token from one place to another.

Denoting the initial state by the associated marking $(2, 0)$, the successive execution of the actions I, II, and III transforms the system into the new state $(1, 1)$. In the associated marked P/T-net, the firing of T causes the same change of the marking.

To manifest the correspondence of the firing mechanism in an arbitrary P/T-net to an enzymatic action in the associated P/T-system, fork and join situation must be explained, too. For this, the firing of an enabled transition in a marked P/T-system is defined to be the state transformation, that arises after completing a specific sequence of enzymatic actions defined bellow.

For this, suppose a P/T-system $\mathcal{S}$ is in the initial state $\Psi_{is}(\mathcal{S})$. The firing of a transition T executes the enzymatic steps $F1, F2, F3, F4$

- $F1$ **: Initial State** $\Psi_{is} \rightarrow$ **Pseudostate** Ψ'
  Every place P in the pre-condition $\bullet T$ of the transition is the target of at least one link starting in M. The transformation $F1$ adds one adjoint link to the set Fund(M, P), if and only if the set Fund(M, P) is empty,

$$\Psi'(\mathcal{S}) \;:=\; F1[\Psi_{is}(\mathcal{S})] = \Psi_{is}(\mathcal{S}) \cup \bigcup_{P \in \bullet T} \{b_P^*\} \quad ,$$

  where b_P^* is the added adjoint at the place P.

- $F2$ **: Pseudostate** $\Psi' \rightarrow$ **Pseudostate** Ψ''
  Motion with trace at M is performed. New fundamental arrows are declared in Fund$(M, P) \circ$ Fund(P, T), if and only if $P \in \bullet T$ and the adjoint b_P^* is present. Since Fund$(M, \mathcal{T})$ is constrained to be unfrustrated by definition, there is exactly one new link from M to T. The links Fund$(\bullet T, T)$ are not modified:

$$\Psi''(\mathcal{S}) \;:=\; F2[\Psi'(\mathcal{S})] = \Psi'(\mathcal{S}) \cup \text{Fund}(M, T) \setminus \bigcup_{P \in \bullet T} \{(b_P^*)^*\} \quad ,$$

  where $(b_P^*)^*$ are the links that have fundamental adjoints in $\Psi'(\mathcal{S})$.

- $F3$ **: Pseudostate** $\Psi' \rightarrow$ **Pseudostate** Ψ'''
  Motion with trace at places in $\bullet T$ removes all added links b_P^*. Since Fund$(\mathcal{P}, \mathcal{T})$ is constrained to be unfrustrated by definition, applying $F3$ transforms the system into the pseudostate Ψ'''

$$\Psi'''(\mathcal{S}) \;:=\; F3[\Psi''(\mathcal{S})] = \Psi''(\mathcal{S}) \setminus \text{Fund}(\bullet T, M) \quad ,$$

  while leaving Fund(M, T) invariant.

- $F4$ **: Pseudostate** $\Psi''' \rightarrow$ **Final State** Ψ_{fs}
  Again, motion with trace at M declares k new links departing from M to places in the post-condition $T\bullet$, where $k = |T \bullet|$, and

$$\Psi_{fs}(\mathcal{S}) \;:=\; F4[\Psi'''(\mathcal{S})] = \Big(\Psi'''(\mathcal{S}) \setminus \text{Fund}(M, T)\Big) \cup \text{Fund}(M, T\bullet) \quad ,$$

  where Fund$(M, T\bullet)$ contains all new links, and

$$|\text{Fund}(M, T\bullet)| \geq k \quad .$$

Applying successively all four steps $F1$ to $F4$, the state if the P/T-system is changed from a state in the state system $\Sigma[\mathcal{S}]$ to another one in $\Sigma[\mathcal{S}]$. Note, that in fig. 3.4 the operation III corresponds to $[F4.F3]$. The firing of a transition in a P/T-system is then defined as follows:

Definition (Firing in a P/T-system):
A transition T enabled by an initial state $\Psi_{is}(\mathcal{S})$ of a P/T-system $\mathcal{S}$, may cause the transformation of the initial state to a (proper) final state $\Psi_{fs}(\mathcal{S})$ by the action sequence

$$F_T \;:=\; [F4.F3.F2.F1]_T \quad ,$$

where the notation means the successive application of the actions $F1$ to $F4$ at the transition T. The transformation is denoted by

$$\Psi_{is}(\mathcal{S}) \stackrel{F_T}{\rightarrow} \Psi_{fs}(\mathcal{S}) \quad ,$$

the transition T is then said to be **firing**.

The definition above applies to every enabled transition in an arbitrary P/T-system $\mathcal{S}$ and does not depend on the number of places in the pre-, or post-condition of the transition. Moreover, from an abstract point of view, the number of places need not even be countable, since the definition of systems does not restrict the set of objects.[2] Although this is not usual for practical considerations, the abstract definition of firing holds in this case, too.

Example

The firing of an enabled transition by execution of the four actions $F1$ to $F4$ is illustrated in the example in fig. 3.5.
The P/T-system in fig. 3.5 consists of five places $P1$ to $P5$ and the transition T. The initial state Ψ_{is} is given by the associated marking

$$m = (1,1,0,0,1) \in \mathbb{N}^5 \quad .$$

Applying the firing transformation F_T on the system $\mathcal{S}$ results in the final state Ψ_{fs} with associated marking

$$m' = (0,0,1,1,2) \in \mathbb{N}^5 \quad ,$$

which is reached via three transient pseudostates from the operations $F1, F2$ and $F3$. This state transformation is identical to the one occurring by the usual firing of T in the associated P/T-net.

The state transformation $\Psi_{is} \stackrel{F_T}{\rightarrow} \Psi_{fs}$ of a transition T in a P/T-system corresponds to a fundamental arrow in the state system $\Sigma[\mathcal{S}]$, which is identical to the link representing the firing of the transition T in the associated marked P/T-net. Since the dynamics caused by a sequence of transition firings in a marked P/T-net is a path in the state system Σ, the following result completes the correspondence of P/T-nets and P/T-systems.

[2] However, here for simplicity it is restricted to small sets.

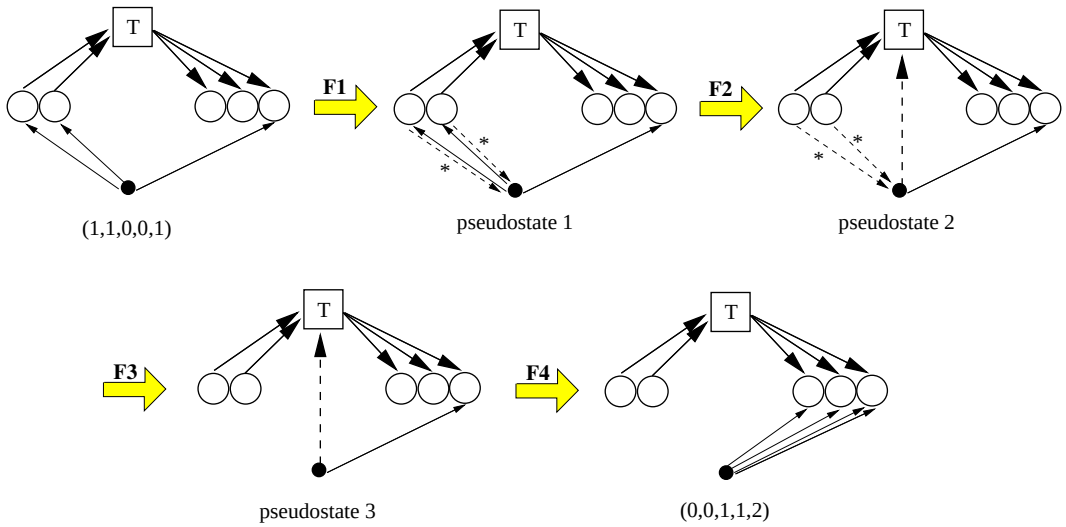


Figure 3.5: The firing of an enabled transition in a P/T-system, realized by elementary enzymatic actions.

Result (P/T-Dynamics):
The dynamics of a P/T-system caused by the action of a sequence of firing transformations F_{T_i} is a path in the state system $\Sigma[\mathcal{S}]$, which coincides with the path from the associated initial marking to the final marking, defined by the same sequence of firing transitions T_i in the associated P/T-net.

The system may evolve automatically, if all enabled transitions are immediately enabled. Then, parallel firing transformations may occur, modeling concurrent systems as in P/T-nets. However, in the systems formulation this mechanism is treated the same way, thus requiring no extensions.

Bypassing inabilities of classical P/T-nets

P/T-nets are a commonly accepted method to model and visualize the basic concepts of the behavior of processes. They enable a very detailed automate-based view of processes. But, like many other low-level modeling techniques, they lack in two main points. First, they do not support advanced composition and refinement mechanisms, meaning that there is no generic way of composing nets to larger ones, or to decompose a given net into smaller pieces. However, these tasks are straightforwardly enabled by formulating the net as a monoid [54], since there a composite P/T-net is simply a colimit of its constituents in an appropriately defined category. Second, the usage of tokens as basic data types lacks abstraction and has limited applicability in modeling more complex systems. Modeling and simulation of complex information systems requires the handling of coupled processes and software modules as 'atomic' entities, that communicate with each other. This means, that

higher-level structures must be used instead of tokens, while the transitions become more advanced operations. Usually, this is possible by using high-level Petri nets, for example *algebraic high-level nets* [52], substituting tokens and markings by algebraic specifications and enabling thereby a more general and abstract data type handling.

The ToCS-based formulation of P/T-nets as P/T-systems bypasses some core inabilities of classical P/T-nets, too. Some advantages have been already mentioned. Yet, there are some further points, that motivate the formulation of P/T-nets in the form discussed above.

- Formulating the P/T-net as a generic system with local dynamics enables the application of any ToCS-based concept. In particular, the composition, refinement and structuring of a P/T-system is based on the ordinary union and decomposition of systems from and into subsystems.

- The firing operation is a local, universal action, that can be easily extended to apply to all immediately enabled transitions in a P/T-system. Therefore, serial and concurrent processes are treated homogeneously.

- The cardinality of the set of places and transitions is not restricted to be finite, or even countable. From a theoretical point of view, the firing operation is well-defined in these cases, too, although not implementable.

- Objects and links in a P/T-system may carry additional structure. For example, the places and transitions may represent any kind of space E_P, E_T, while relations to and from a transition may be mappings u_{PT}, or u_{TP} between these spaces. Allowing loops l_T at the transition T, one gets mappings

  $$u_{TP'} \circ l_T \circ u_{PT}$$

  between places. The state of the system Ψ is described by a point in the product of the spaces E, that is specified from pairs $(M, m_P(M))$ with $m_P(M) \in E_P$, represented by a marking. In this sense, it is possible to handle more complex data types. In particular, weighted P/T-nets are simply represented as P/T-systems, where every link between places and transitions carries a number in $\mathbb{Z}$, which is added or subtracted in a firing operation, depending on the direction of the link.

- Using ToCS-based formulations of various modeling approaches, allows their combination into hybrid approaches. This is an essential step towards the treatment of complex software modules as black-boxes (objects) in dynamical networks. For example, since places are objects, they may possess internal structure, that describes their internal reactions on receiving signals. This means, that the place may be replaced by a state-transition system. A receiving signal may be the declaration of a new fundamental arrow, whose target the considered place is. Hybrid approaches may enable an integrate modeling of various views of a system.

- As a more advanced feature, one may try to analyze the structure of a P/T-system enzymatically. In addition, one may apply enzymes, to modify the structure. For example, one may identify all places in the pre-condition of a transition, constructing thereby a coarser view of the system.

However, as argued in the next chapters, the complexity of a system requires a rather different modeling paradigm.

3.3 PHYSICAL LAWS AND SYSTEMS THEORY

Although the theory of complex systems enables the treatment of systems arising in various disciplines, it is of great interest to analyze, to what extend it is possible to apply the theory in the description of well-known physical phenomena. More than just being one typical class of systems, physical systems may help both, to justify the formal analytical framework developed so far, and to serve for feasible examples for the dynamics of a system. To illustrate this treatment, the following subsections will investigate two examples for physical systems, namely classical electrodynamics and the Dirac equation.

Usually, the system theoretical description of a physical theory starts with a discretized version of the continuum theory. We define the nodes of the discretization grid to be the objects x of the system representing the discretized space. The links of the grid are defined to be the fundamental arrows $b_i(x)$ of the space system, where i numerates in some way the links with source-object x. A system that occurs in this way is a unfrustrated complete system, the adjoint operation is defined as

$$b_i(x)^* = {b_i}^{-1} = b_{_i}(x + \vec{i}) \quad ,$$

where the grid is assumed to be regular and $b_i(x) : x \to x + \vec{i}$.

To define the physical system suppose the fields of the theory are the elements of an appropriately defined space V equipped with an internal (not necessarily commutative) composition law and a $*$-operation. The physical system is then constructed from this data by defining its objects to be the space points x and its arrows to be generated by elements $v \otimes b_i(x)$ where $v \in V$ and $b_i(x)$ a fundamental arrow of the grid and the $\otimes$-sign symbolizing a formal product. The composition of arrows is canonically defined to be

$$\Big(w \otimes b_j(y)\Big) \circ \Big(v \otimes b_i(x)\Big) = wv \otimes b_j \circ b_i \quad ,$$

with composable b_i , b_j. The adjoint operation is defined according to

$$\Big(v \otimes b_i(x)\Big)^* = v^* \otimes b_{_i}(x + \vec{i}) \quad .$$

The general system theoretical gauge transformations defined in (2.4.2) take the concrete form

$$v \otimes b_i(x) \to g_y v g_x^{-1} \otimes b_i(x) \quad ,$$

with gauge fields $g_x \in V$ corresponding to loops $g_x \otimes \iota_x$. Typically V will include the parallel transporters of the fields in the theory (lattice gauge theory, electrodynamics, etc.).

The dynamics of the system are given from the universal dynamics laws for the fundamental arrows $v \otimes b_i(x)$. The system corresponds to the considered physical theory if the universal dynamics reproduce the equations of the theory for the quantities $v \in V$. The task is then to find a finite operation that respects system locality while realizing the dynamical evolution of the equations of motion of the theory. Since many differential equations in physics are second order equations that corresponds to pairs of canonical variables, the arrows of the system are defined to consist of two different types of arrows in these cases. Consequently this picture weakens the uniqueness of the universal dynamics law by mixing the different types of links. This is the case in classical electrodynamics considered below, whereas the system theoretical Dirac equation does not need this construction.

3.3.1 CLASSICAL ELECTRODYNAMICS

In order to formulate a system theoretical analogon of classical electrodynamics, the Maxwell equations are briefly expressed here based on the electric field E^a and the vector potential A_b as basic variables. Once vector potentials are introduced, the only constraint of the theory remains the Gauss-constraint

$$\int_V d^3x \; \partial_a E^a \; = \; Q_V \quad , \tag{3.3.1}$$

where Q_V is the amount of electric charge in the finite spatial volume V, the index a is a spatial index.
To simplify the notation, the scalar potential Φ is eliminated by the gauge transformation

$$\begin{aligned} A_a \quad &\rightarrow \quad A'_a := A_a + \partial_a \int dt \, \Phi \\ \Phi \quad &\rightarrow \quad \Phi' := 0 \quad , \end{aligned} \tag{3.3.2}$$

where the primes are suppressed in all further equations.

In this gauge, the dynamical equations are given by the Maxwell equations [55, 56]

$$\delta^{ab} \dot{A}_a \;\; = \;\; -E^b \tag{3.3.3}$$

$$\dot{E}^a \;\; = \;\; \epsilon^{abc} \partial_b B_c - J^a \, , \tag{3.3.4}$$

with B being the magnetic field of the potential A

$$B = \nabla \times A$$

and J the current satisfying the continuity equation

$$\int_V d^3x \; \partial_a J^a = \dot{Q}_V \, . \tag{3.3.5}$$

The general form of equation (3.3.3) with non-vanishing scalar potential is then achieved by inverting the gauge transformation (3.3.2).

ToCS-based formulation

To turn over to the system theoretical formulation, the theory is discretized on a regular 'flat' lattice. Discretizations of the Maxwell equations were considered for example in [57, 58]. Here the discretization is assumed to be performed on a hexagonal space lattice corresponding to a face-centered cubic lattice. The lattice constant a is assumed to be small compared to all physical variables, $a << 1$. This enables the approximation of theory entities by expansions in a. The objects x are given by the nodes of the lattice, i.e. the discretized space points, and the links $b_l(x)$ are given by the edges from point x to $x + \vec{l}$, where $l = \pm 1, \pm 2, \ldots$ enumerates a link-direction and its opposite direction at an object and $\vec{l}$ is the corresponding vector in direction l with $|\vec{l}| = a$. This space system is a complete system, its adjoint operation is simply the inversion of the link direction. An object together with its local neighborhood is shown fig. 3.6.

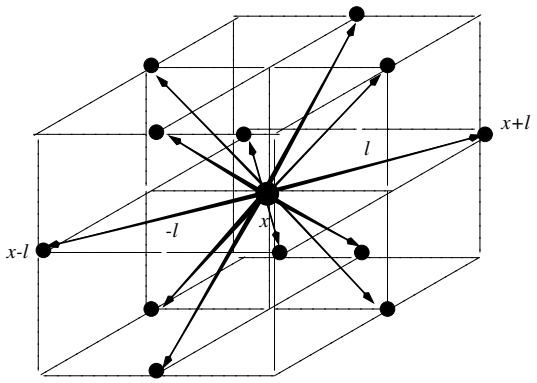


Figure 3.6: The local structure of the face-centered cubic grid. Opposite links are labelled by a minus-sign.

As described above the arrows of the physical system will be generated by formal products of expressions of the fields E, A, J, Q with the links of the space system. Given the objects and links of the physical system, its semantical content is defined by assuring the validity of certain theory specific constraints for the system. Further, to achieve the commutative gauge group $U(1) \cong \exp(\imath\mathbb{R})$ of electrodynamics, a typical link of the physical system is defined to be of the form

$$e^{\imath v^l} \otimes b_l \quad ,$$

with $v^l = (\vec{v}, \vec{l})$ being the component of a field v in direction of the considered link. Assuming the lattice constant a to be small, the ordinary parallel transporters U of the theory along a link b_l from an object x to $x + \vec{l}$ are given by

$$\begin{aligned} U_l(x) &:= \mathcal{P} e^{-\imath \int_b A_a \, dx^a} \otimes b_l(x) \\ &= e^{-\imath a A_l(x+\vec{l}/2)} \otimes b_l(x) \quad , \end{aligned} \tag{3.3.6}$$

where $\mathcal{P}$ means path ordering of the product. The path ordering is neglected with respect to the lattice constant, the vector potential is approximated by its value in the middle of the link. The parallel transporters are represented graphically by thin arrows.

The 'conjugate momenta' P of the parallel transporters U are links defined as

$$P_l(x) := e^{-\imath a A_l} e^{-\imath a \tau E^l(x)} \otimes b_l(x) \quad , \tag{3.3.7}$$

where the electric field is considered at the object x. As will be shown below, in order to reproduce the Maxwell equations the time step τ must be related to the geometry of the lattice. The links P are graphically represented by thick arrows.

The boundary conditions of the theory may be expressed for example by the current $J(t, x)$ given for all times and the charges Q_x at initial time. The current is implemented according to its geometrical meaning as links I in the physical system

$$I_l := e^{-\imath a A_l} e^{-\imath a \tau^2 J^l(x)} \otimes b_l(x) \quad , \tag{3.3.8}$$

with adjoints

$$I_l^* = e^{+\imath a A_l} e^{+\imath a \tau^2 J^l(x)} \otimes b_{-l}(x + \vec{l}) \quad , \tag{3.3.9}$$

whereas the charge Q_x is assumed to be located at the space point x and thus being represented in the system by the loop S_x at x in the form

$$S_x := e^{\imath a \beta \tau Q_x} \otimes \iota_x \quad , \tag{3.3.10}$$

where the geometric factor $\beta = N/O$ comes from the discretization of the theory. It contains the boundary area O of the figure builded around an object with its N neighbours as corners. For a two-dimensional triangular lattice $\beta = a^{-1}$, for a three dimensional face-centered cubic lattice $\beta = \sqrt{3}/(2a^2)$. The links I are represented graphically as dashed arrows, the loops S as dashed loops.

The charge $Q_x^{t+\tau}$ a time step later is assumed to be determined by the continuity equation. Its discretized version used here is given by

$$\sum_l E_l = \beta \, Q_V \quad . \tag{3.3.11}$$

The continuity equation (3.3.5) is then simply stated system theoretically in the form

$$S_x^{t+\tau} = S_x^t \circ \prod_l \left(I_{-l}(x + \vec{l}) \circ U_l(x) \right) \ , \tag{3.3.12}$$

that is, expressing the charge distribution a time step later by the current J. The product on the right hand side of (3.3.12) represents the successive non-commutative composition of arrows in the system. The product is over all fundamental arrows with source x. Expanding the product on the right hand side of (3.3.12) one results with the sum of the current $\sum_l J^l$ on the lattice and thus the continuity equation (3.3.11).

To complete the specification of the static system the Gauss-constraint needs to be formulated system theoretically too. It is expressed by the constraint

$$\prod_l \left(P_{-l}(x + \vec{l}) \circ U_l(x) \right) = S_x \ , \tag{3.3.13}$$

which must be valid for all times. This equation is the exponentiated version of (3.3.1) since

$$P_{-l}(x + \vec{l}) \circ U_l(x) = e^{\imath a \tau E^l(x+\vec{l})} \otimes \iota_x$$

leads to the discretized divergence of the electric field $(\nabla \cdot E)$ at x on the lattice. Graphically equation (3.3.13) is stated in fig. 3.7.

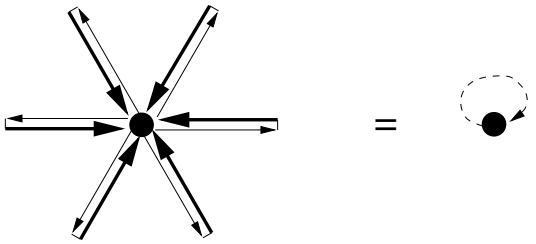


Figure 3.7: Graphical form of the Gauss-constraint. Thin links represent the parallel transporters U, the thick arrows stand for the links P. The dashed loop on the right hand side symbolizes the loop S_x, the charge localized at the object. The figure is exact in two dimensions for a triangular grid, for a three-dimensional face-centered cubic lattice six links not lying in the plane above must be added.

The dynamical equations for the fields E and A correspond to the time evolution of the arrows U and P in the system. Both, the time evolution of U and the one of P follow a universal dynamics law expressing the arrows at a given time by arrows of the system a time step earlier. As already mentioned, the dynamics will mix both types of links in the expressions for the links after a time step. To result with the Maxwell equation (3.3.3) the universal dynamics of the parallel transporters U is given by

$$U_l^{t+\tau} = U_l^t \circ P_l^{t*} \circ U_l^t \ , \tag{3.3.14}$$

since expressing both sides yields

$$\begin{aligned} e^{-\imath a(A_l+\tau\dot{A}_l)} \otimes b_l &= U_l \circ P_l^* \circ U_l \\ &= e^{-\imath a(A_l-\tau E^l)} \otimes b_l \ , \end{aligned}$$

which is the exponentiated equation (3.3.3). Due to the definition of the arrows P in (3.3.7) equation (3.3.3) expresses that the new thin links 'stores' the electric content of the old thick ones. In graphical notation this is seen in fig. 3.8.

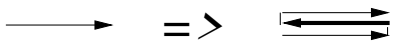


Figure 3.8: Universal dynamics of the parallel transporters U represented by thin links. Due to the mixing with the thick links, the parallel transporters save the electric field a time step earlier.

In a similar way the universal dynamics law for the arrows P is given by the equation

$$P_l^{t+\tau} = (U_l^t \, I_l^{t*} \, U_l^t) \circ \prod_{\triangle_l} U^t(\triangle_l) \ , \tag{3.3.15}$$

where $\triangle_l$ is an elementary triangle at b_l, i.e. a loop consisting of three different links, which contains the link b_l. The product is over all triangles having b_l in common and $U(\triangle_l)$ is the associated parallel transporter around the loop as shown in fig. 3.9 in two dimensions.

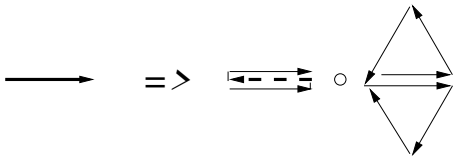


Figure 3.9: Universal dynamics of the thick links P. The dashed arrow is the adjoint of the 'current' link I. In three dimensions another pair of triangles lying in a plane must be added.

In the special case where no sources ($Q_x = 0$) and no currents ($J^l(x) = 0$) are present the links I and the loops S are given by

$$I_l(J=0) = U_l \tag{3.3.16}$$

$$S_x(Q=0) = \imath_x \ , \tag{3.3.17}$$

as is seen from their definitions (3.3.8) and (3.3.10). The equation for the thin links is not affected and remains in the same form, whereas the dynamics equation (3.3.15) for thick links P becomes

$$P_l^{t+\tau} = U_l^t \circ \prod_{\triangle_l} U^t(\triangle_l) \ , \tag{3.3.18}$$

in agreement with the results in [31].

To prove that (3.3.15) yields the Maxwell equation (3.3.4) for the E field, the product on the right hand side is expanded. To simplify the calculation,the grid is restricted to a two-dimensional triangular lattice, the generalization to a three-dimensional face-centered cubic lattice is more or less straightforward. In two dimensions the triangular lattice possesses exactly two triangles containing a link b_i, they are written $\triangle$ and $\triangledown$ representing the two triangles in fig. 3.9. The product in (3.3.15) is then calculated as follows

$$\begin{aligned}
\prod_{\triangle} U^t(\triangle) &= U(\triangle) \circ U(\triangledown) \\
&= \Big(U_{-j}U_{-k}U_i\Big)\Big(U_{-k}U_{-j}U_i\Big) \\
&\approx e^{\imath\frac{\sqrt{3}}{4}a^2\partial_y A_x(y_+)} e^{-\imath\frac{\sqrt{3}}{4}a^2\partial_y A_x(y_-)} \otimes \iota_x \\
&= e^{\imath\frac{\sqrt{3}}{4}a^2(B_z^- - B_z^+)} \otimes \iota_x \\
&\approx e^{-\imath\frac{\sqrt{3}}{4}a^2(\frac{\sqrt{3}}{2}a\partial_y B_z)} \otimes \iota_x \\
&= e^{-\imath a\gamma(\nabla\times B)_i} \otimes \iota_x \ ,
\end{aligned}$$

with $\gamma = 3/8\, a^2$. The factors originate in the discretized derivative. The curl operators are considered as generic two-dimensional, that is, as a plaquette property. Setting the time step to be $\tau^2 = \gamma$ and thereby depending on the lattice geometry , equation (3.3.15) leads to

$$e^{-\imath a(A_i + \tau^2 \dot{E}^i)} \otimes b_i = e^{-\imath a(A_i - \tau^2 J^i + \tau^2(\nabla\times B)_i)} \otimes b_i \ ,$$

which is the Maxwell equation (3.3.4).

To generalize this result to a three-dimensional face-centered cubic grid a basis is introduced at the considered link b_l. Since every link is part of four independent elementary triangles lying as pairs in two linear independent planes which intersect at the line generated by b_l, the basis $(\vec{e}_x, \vec{e}_y, \vec{e}_w)$ is defined by $\vec{e}_x = b_l$ and the two further unit vectors lying respectively in the two planes of the triangle pairs (fig. 3.10). Due to the geometry of the lattice this coordinate system is not orthogonal. Nevertheless, in the product (3.3.15) each pair of opposite triangles leads to a difference of the magnetic field component orthogonal to the triangle plane in the middle of the triangles. Taking account also the orientation of the triangles, this yields the sum $[(\sqrt{3}/4)a\sum B_n]$ of the magnetic field at the four triangle middle points. Approximating Stokes' theorem for the loop in fig. 3.10 results in

$$\sum_{n=1}^{N} B_n = \frac{NO}{L}\,(\nabla \times B)_x \quad ,$$

where $N = 4$ and O is the surface and L the length of the loop. Defining the factor γ to be

$$\gamma = \frac{\sqrt{3}Oa}{L} \ ,$$

that is, proportional to a^2, the universal dynamics law (3.3.15) applies also for the links in a three-dimensional face-centered cubic lattice when setting $\tau^2 = \gamma$.

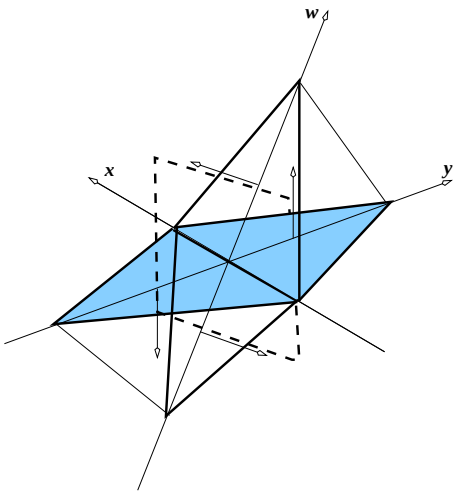


Figure 3.10: The two pairs of elementary triangles containing a common link. The dashed loop lying in the yw-plane is approximated to produce the curl of the magnetic field along the x-axis.

The general form of equation (3.3.4) is achieved by a gauge transformation of the system. As defined in the general framework the $U(1)$-loops $L_x := e^{-\imath a \Lambda(x)} \otimes \iota_x$ are introduced, where $\Lambda = -\int dt \Phi$ is the gauge function leading to the initial scalar potential Φ. The links of the system transform according to the definition described in the previous sections, the parallel transporters become

$$\begin{aligned} U_l'(x) &= e^{-\imath a A_l'} \otimes b_l \\ &= L_{x+\vec{l}} \circ U_l(x) \circ {L_x}^{-1} \\ &= e^{-\imath a A_l + a\,\partial_l \Lambda} \otimes b_l \ , \end{aligned}$$

which is the ordinary transformation law for the vector potential. Gauge transforming the thick links, too, leaves the electric fields invariant, leading thereby to the general equation

$$\dot{A} = -E - \nabla \Phi$$

for the time evolution of the vector potential in presence of a non-vanishing scalar potential. Summarizing the results above, the following table shows the system theoretical formulation of the Maxwell equations in a compact fashion.

Continuum	ToCS-based
$\int_V \partial_a E^a = Q_V$	$\prod_l \left(P_{-l} U_l \right) = S_x$
$\delta^{ab} \dot{A}_a = -E^b$	$U_l^{t+\tau} = U_l^t P_l^{t*} U_l^t$
$\dot{E}^a = \epsilon^{abc} \partial_b B_c - J^a$	$P_l^{t+\tau} = (U_l^t \, I_l^{t*} \, U_l^t) \prod_{\triangle_l} U^t(\triangle_l)$
$\int_V \partial_a J^a = \dot{Q}_V$	$S_x^{t+\tau} = S_x^t \circ \prod_l \left(I_{-l} U_l \right)$

Some examples are presented in the following.

Examples

(i) To illustrate the approach, electrodynamics in vacuo is considered as a simple example. In this case the current and the charges vanish, $Q = J = 0$, leading thereby to links S and I given by (3.3.17). The most simple example would be to consider the electrostatics of a constant homogeneous electric field $\vec{E} = E \, \vec{i}$. This case is easily calculated and behaves consistent with the considerations above, that is, the system is unfrustrated with respect to the thin links U and evolves according to

$$\begin{aligned} U_t{}^* \circ U_{t+\tau} &= e^{\imath a \tau E} \otimes \iota_x \\ P_t{}^* \circ P_{t+\tau} &= e^{\imath a \tau E} \otimes \iota_x \quad . \end{aligned} \tag{3.3.19}$$

Since the electric field remains constant for all times no further significant dynamics is performed.

(ii) Another simple example is to consider plane electromagnetic waves in vacuo. In this case the periodic change of the fields should be reproduced by the universal dynamics defined in (3.3.15) and (3.3.14). To see this, consider the two-dimensional case with the electric field $\vec{E}$ lying in the $\vec{i}$-direction of a triangular grid (the x-axis of the coordinate system) and the wave vector $\vec{\kappa}$ lying perpendicular to this axis in the grid plane (the y-axis of the coordinate system), yielding thereby a $\vec{B}$ field orthogonal to the grid. The fields are defined as

$$\begin{aligned} \vec{E} &= E \cos(\kappa y - \omega t) \vec{i} \\ \vec{A} &= \omega^{-1} E \sin(\kappa y - \omega t) \vec{i} \\ \vec{B} &= -\frac{\kappa}{\omega} E \cos(\kappa y - \omega t) \vec{e}_z \\ \nabla \times \vec{B} &= \frac{\kappa^2}{\omega} E \sin(\kappa y - \omega t) \vec{i} \; , \end{aligned} \tag{3.3.20}$$

with $c = 1$ and $\kappa^2 = \omega^2$. From (3.3.20) the links of the system are determined to be

$$U_i = e^{-\imath a \frac{E}{\omega} \sin(\kappa y - \omega t)}$$

$$\begin{aligned} U_j = U_k &= e^{-\imath a \frac{E}{2\omega} \sin(\kappa y - \omega t)} \\ P_i &= e^{-\imath a E(\omega^{-1} \sin(\kappa y - \omega t) + \tau \cos(\kappa y - \omega t))} \\ P_j = P_k &= e^{-\imath a \frac{E}{2}(\omega^{-1} \sin(\kappa y - \omega t) + \tau \cos(\kappa y - \omega t))} \quad . \end{aligned} \tag{3.3.21}$$

The universal dynamics of the thin links U are given by

$$\begin{aligned} U_i^{t+\tau} &= e^{-\imath \frac{aE}{\omega} \sin(\kappa y - \omega t - \omega\tau)} \\ &\approx e^{-\imath \frac{aE}{\omega} \sin(\kappa y - \omega t) + \imath \tau a E \cos(\kappa y - \omega t)} \\ &= U_i P_i^* U_i \quad , \end{aligned} \tag{3.3.22}$$

up to terms in τ^2. For the universal dynamics of the thick links P the parallel transporters around the two elementary triangles containing the common link U_i are calculated

$$\begin{aligned} U(\triangle) &= U_{-j} U_{-k} U_i \\ &= e^{-\imath a \frac{\sqrt{3}}{4} a B_z^+} \end{aligned}$$

and

$$\begin{aligned} U(\triangledown) &= U_{-k} U_{-j} U_i \\ &= e^{\imath a \frac{\sqrt{3}}{4} a B_z^-} \quad , \end{aligned}$$

where B_z^+ is the magnetic field in the middle of the upper triangle and B_z^- the magnetic field in the middle of the lower one. Building the product of both triangles and another U_i as in (3.3.15) yields

$$\begin{aligned} U_i U(\triangle) U(\triangledown) &= U_i e^{-\imath a \gamma \partial_y B_z} \\ &= e^{-\imath a \omega^{-1} E \sin(\kappa y - \omega t)(1 + \gamma\kappa^2)} \\ &= P_i^{t+\tau} \quad , \end{aligned}$$

with $\tau^2 = \gamma$ confirming the universal dynamics law (3.3.15).

3.3.2 THE DIRAC EQUATION

As a further application, the ToCS-based formulation the Kogut-Susskind discretized version of the Dirac equation. Since this is a first-order equation, only one type of arrows is present in the system. The universal dynamics law for the Dirac equation is similar to the universal dynamics of the thick links P in the ToCS-based formulation of classical electrodynamics.

To enable the demanded formulation of the Dirac equation, we make use of the Kogut-Susskind discretization [59], which is considered in the following form

$$\psi(X, t+\tau) = \psi(X, t) \; + \; \tau \sum_{k=\pm 1, \pm 2, \pm 3} \eta^{-k}(X + \vec{k}) \; \psi(X + \hat{k}, t) \quad , \tag{3.3.23}$$

where ψ is a two-component spinor, and η denotes the parallel transporters on the lattice.

The equation (3.3.23) corresponds to a universal dynamics law, that may be presented in graphical form, as shown in fig. 3.11

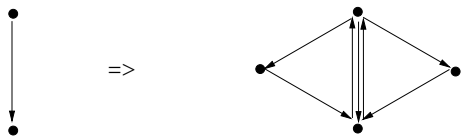


Figure 3.11: The universal dynamics of the Dirac equation.

The details of the construction of an appropriate system $\mathcal{S}_D$ and the universal dynamics are found in appendix B. It is based on a regular, three-dimensional space-lattice Γ with vertices $V(\Gamma)$, which is extended by one additional point Ω_∞. The objects of the system are given by state spaces Ω assigned to every site of the extended lattice with additive functions between them. The link from a point $X \in V(\Gamma)$ to the additional point is given by the Dirac field $\psi(X)$ the adjoint link by its conjugated field ψ^*. The arrows are given by sections $\mathcal{C}[P]$ and $\mathcal{B}[P]$ in a specific bundle B with base P the free paths in the extended lattice. The fibre in $f \in P$ is given by the set of subpaths of f considered as additive functions. The section $\mathcal{B}$ is a simple embedding of P into B, whereas the section $\mathcal{C}(f)$ includes the sum of all irreducible subpaths of f. Irreducible subpaths are subpaths, which are given by extracting loops from the original path f. They are precisely defined in appendix B. In the bundle B a natural composition law $\odot$ exists, for $p, q \in B$

$$p \odot q \;:=\; \Big(f \circ g \;,\; \sum_{\xi \in I_\alpha(f \circ \chi)} \tilde{\xi} \quad \Big) \quad ,$$

where $p = (f, \tilde{\phi})$ and $q = (g, \tilde{\chi})$ are elements in the bundle with composable projections $\pi(p)$ and $\pi(q)$, ϕ and χ are elements of the corresponding fibres.

A constraint must be imposed on the set of arrows to enable the Dirac equation. It is given in the following form

$$\mathcal{B}\Big(\prod_i b_i \Big) = \odot_i \; \mathcal{C}(b_i) \quad , \tag{3.3.24}$$

for reducible paths of fundamental arrows b_i of the extended lattice. It is important, that the section $\mathcal{C}$ is a homomorphism with respect to the composition laws

$$\mathcal{C}(f \circ g) = \mathcal{C}(f) \odot \mathcal{C}(g) \quad .$$

The construction of the system $\mathcal{S}_D$ is summarized in the following result:

Result (Dirac-System):
The Dirac-System $\mathcal{S}_D$ consists of objects

$$Ob(\mathcal{K}_D) \;\cong\; V(\Gamma) \cup \{\Omega_\infty\} \;\;, \tag{3.3.25}$$

where Γ is a regular three-dimensional lattice. The arrows of the system are given by the union $\mathcal{C}[P] \cup \mathcal{B}[P]$ of the sections $\mathcal{C}$ and $\mathcal{B}$ and are generated with respect to the constraint (3.3.24) by the fundamental arrows $\mathcal{C}[Fund(\mathcal{K}_D)]$ and the formal identities at each object.

As shown in the appendix, applying the constraint (3.3.24) to specifically constructed paths leads to the Dirac-dynamics. The paths include the loops $l_k : \Omega_\infty \to \Omega_\infty$, which are defined as follows

$$l_k := \psi^* \circ u_k \circ \zeta_k \;\;,$$

where $\zeta_k : \Omega_\infty \to \Omega_{Y_k}$ and $u_k : \Omega_{Y_k} \to \Omega_X$ are paths that do not intersect each other outside their common object Ω_{Y_k}. The k numerates the six lattice-neighbors of a lattice point Ω_X. Then the dynamics are given by the following result:

Result (Dirac-Dynamics):
In the Dirac system $\mathcal{S}_D$, the universal dynamics law

$$\psi(X, t+\tau) = \psi(X) \circ \prod_{k=\pm1,\pm2,\pm3} l_k(X) \;\;, \tag{3.3.26}$$

corresponds via the constraint (3.3.24) to the Kogut-Susskind discretization of the Dirac equation with discretized time $t \in \tau\mathbb{Z}$.

As follows from the explicit construction in the appendix, the results above implement the minimal condition to include a universal dynamics corresponding to the Dirac equation. However, the constraint (3.3.24) may further restrict the system.

3.4 SUMMARY

In this chapter the modeling approach based on the ToCS-framework is introduced and related to some well-known modeling techniques. It is shown how, using the same concepts and methods, both physical systems, and information systems may be analyzed in a ToCS-based formulation. The investigations focus on the structure of the considered systems, as well as on their dynamics. In particular, for information systems, views, graph transformations, state-transition systems, and Petri nets are considered, whereas the physical systems examples are classical electrodynamics and the discretized Dirac equation for massless fermions. In this sense, the applicability of the ToCS-based concepts in rather different contexts, corroborates its universal character.

Chapter 4

COMPLEX INFORMATION SYSTEMS

"The source of all confusions is the misleading word 'object'."
L. Wittgenstein in 'Philosophical Grammar'

There are many directions one may choose, when trying to extend the scope of complex systems theory. Nevertheless, the underlying network structure requires the definition of precise model elements. This means, that both, objects, and their relations must be well-defined. Although many systems possess a natural system structure, a simple modeling, based on a network, may still be very unsatisfactory. The reason is, that in many systems, both, the internal structure of the objects, and the concrete form of their relations may be too complex and extensive to be appropriately modeled by an elementary system structure. This is the case for sociological, or economical systems, although some important investigations in both areas are possible from an systemic point of view [25, 15].

To overcome these inabilities, one may concentrate on systems, which enable the definition of objects with feasible internal structure, as well as precise interactions, based on a set of assessable actions. These conditions are inherently fulfilled in software systems and in some more general information systems, since the need for executable systems requires the precise specification of the system structure and behavior. However, the simple modeling techniques of the previous chapter, are not sufficient to incorporate the whole structure and behavior of an information system. Even if some elementary processes in an information system may be described by low-level techniques, the analysis of a system as a whole may be infeasible. Fortunately, systems underlying the object-orientation paradigm possess a generic network structure and offer the ability of systemic investigations. Therefore, they form a distinguished area for investigations in complex systems.

After illustrating the role of software intense systems in the investigation of complex systems, aspects concerning their internal complexity are discussed. It is argued, that a highly conceptual modeling approach may increase the system lifetime considerably. The basic ideas of object-orientation and some of the recent methods to handle system engineering

within this approach are explained. Since object-orientation considers information systems as *structured networks of objects* [60], it shares some essential ideas with the system theoretical approach and allows a formulation of its fundamental ingredients within the ToCS-framework. Also, many other modeling approaches allow a object-oriented formulation, like object-oriented Petri nets [61], or object-oriented state-transition systems [62].

The usage of object-orientation is in particular motivated as a foundation for specifying the Unified Modeling Language (UML) modeling approach in the next chapter. Since the UML is based on a metamodel that is defined with respect to object-oriented concepts it is necessary to introduce the main ideas of this paradigm. Although the UML is described in detail in the next chapter this chapter contains a short description of the view-architecture of the UML.

4.1 COMPLEXITY IN INFORMATION SYSTEMS

Information systems enable computer simulations and are widely used to analyze, illustrate and handle the static and dynamical properties of complex systems. While very essential, this does not manifest the only interplay between software and complex systems. Here, software intense systems are considered from a somewhat opposite point of view, focusing on their creation, structure and evolution as a dynamical system. In particular, it is described how the relevant constituents such systems possess can be handled in appropriate models.

The process of software creation includes its specification, design, development and testing as well as maintaining and modification of existing material. What makes software intense systems really tough during this process is the interaction between its components, which can take almost any complicated form. Modifying one module of the system could affect nearly every other part of it, change data used elsewhere or cause a hidden problem, like overwriting memory in a worst case. Also a modified transaction will impact any module of the system which uses it, and whether or not it works depends on the context of the usage. Since this is a relevant feature for commercial products, various approaches to structure the used software have been developed in the past [63]. However, this type of behavior also characterizes general complex systems.

Our interest in information systems arises therefore in a twofold way. First, they are complex systems themselves, interacting with other information systems or with external users in a precise fashion. This means that they can serve as examples for complex systems with environment. Second, since a tremendous amount of work has been investigated to develop methods to treat their complexity, one can learn from the various existing modeling approaches about the relevant issues and how they have to be handled. The concept of abstract data types, unifying data and corresponding functions into modular units, is one major example, leading to object-orientation, which will be discussed later on. Still more conven-

tional techniques, like data flow diagrams showing the way information is propagated in the system and therefore its dynamics, are useful ideas to understand dynamical properties.

As argued above, the complexity of information systems is a measure for their internal interactions and mutual dependencies. Usually, it increases the larger the system becomes and the higher its initial complexity was. "Natural" complex systems usually underly an evolution due to internal or external procedures, for example genetic evolution of biological systems, or evolution into equilibrium states in thermodynamical systems. This is true also for information systems. They have to be optimized with respect to their developers (internal demands) and to their users (external demands):

internal demands	**external demands**
easy to modify and extend	correct and quick results
easy to understand and test	reliability
high re-usability	easy to use and learn
compatible, portable	effective

This continuous modification of a system yields a dynamical evolution of the system's structure. In analogy to thermodynamical systems such a dynamical evolution may heuristically be expressed by a kind of increasing entropy (complexity) of the information system. Heuristically, the evolution of information systems can be described by a number of laws, two of which were according to Lehman and Belady [64]:

I. Used software will be modified.
II. Modifications of the software will increase its complexity, provided that one does not actively work against this.

Suppose the complexity of a system is given by a function C measuring the intensity of the interconnections of different parts of a system. Such a function may be given for example by the average number of possible ways a system module can affect other parts of the system, that is, by the average number of possible data flows starting at a fixed source. Another possible measure for the complexity of a system structured in form of a communication network is the *McCabe cyclomatic complexity metric*, defined for graphs [45]. The idea is to represent all possible sequences a program can take as a graph with n nodes and l directed connections. The McCabe cyclomatic complexity metric M is then given by

$$M = l - n + 2 \quad . \tag{4.1.1}$$

It is easy to calculate the McCabe cyclomatic complexity for ordered systems. For a linear system with n nodes and each two successors connected by one (simple) or respectively

two (bidirectional) links one gets

$$\begin{aligned} M[\text{simple}] &= 1 \\ M[\text{bidirectional}] &= n \quad , \end{aligned}$$

whereas for a unfrustrated, bidirectionally connected system (every object being connected to all others uniquely) one gets a quadratic dependence

$$M[\text{biconn., unfrust.}] = (n-1)^2 + 1 \quad .$$

Usual standards require a low McCabe number, an isolated elementary operation should possess a number less than 10. Due to polymorphism, the McCabe number is sometimes ill-defined and therefore of less interest in object-oriented approaches, although it may be used to restrict use-cases. [1]
However, according to the heuristic laws stated above we expect an increase of the system complexity C by all usual modifications. Further, experience shows that it is reasonable to assume, that the increase of complexity depends on its initial value. Since it is much easier to modify an ordered system without destroying its order, a simple assumption would be to expect the change to be proportional to its initial value, yielding an exponential dependence of the complexity in time

$$\begin{aligned} \dot{C} &= \alpha_M C \\ \Rightarrow \quad C &= C_0 \, e^{\alpha_M t} \end{aligned} \tag{4.1.2}$$

The coefficient α_M is a parameter describing the quality of the modeling approach as its average ratio of affected system parts in time resulting from an elementary modification of an isolated module. Modeling approaches that do not reflect the logical structure of the system, enabling for example every type of interaction between all parts of the system (like most procedural programming languages do), are assigned a high α_M (fig. 4.1).

To get strongly structured models and to avoid "illegal" interactions, large information systems are usually developed with the help of so called CASE-tools (Computer Aided System Engineering), so that in essence the modeling parameter α_M describes both, the quality of the underlying methodology and of its implementation in the used tool. The more structure the CASE-tool reflects, that is, if it describes the logical and the physical structure of the system and handles all their possible interactions in a well-defined and precise way, the smaller its parameter will be, resulting in a slower increase of their complexity in time (fig. 4.1).

To understand the way a conceptual modeling approach may help to reduce the complexity of a system, let us state some essential requirements it must fulfill. First, to be able to specify a problem scenario, it must possess a modeling language, enabling the analysis and

[1] Use-Cases define typical applications of the system and may contain other use-cases as internal structure. This internal structure may then be restricted by the McCabe number.

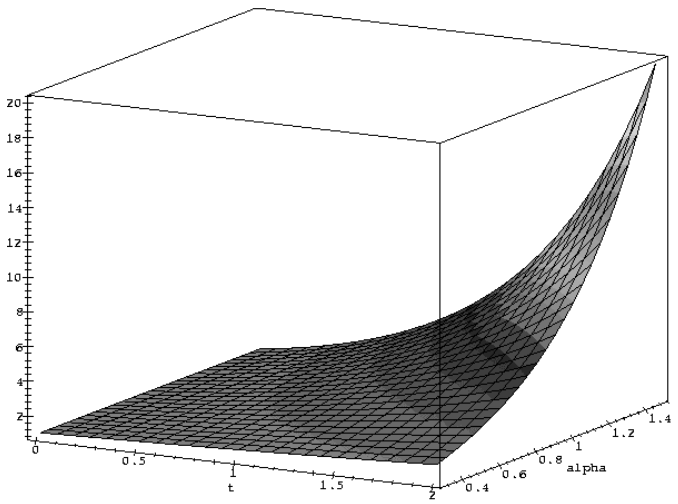


Figure 4.1: Exponential growth of the complexity of information systems considered as a function of time and modeling parameter α_M, which describes the quality of the used modeling approach.

representation of the modeled system to any desired detail. Since such an analysis will usually lead to a highly interconnected network model, a visual language, independent of the future implementation, is desirable. To succeed in constructing the system, a clear syntax and well-formedness rules must be defined for the used language. Further, the documentation of the system up to a certain degree of detail is demanded. For example, documentation issues require the system elements to carry unique specifications (names).

Conceptual modeling is realizable best within the context of an integrate development environment, governed by a method for the analysis and a development process built upon this method and supported by CASE-tools. The defined language enables the representation of the system, with respect to the used method. A corresponding CASE-tool would then not only aid the translation of the constructed model into program code but also the *reengineering* of code into the used modeling notation. This would reduce the exponentially increased system complexity resulting from direct modifications of the program code. The evolution of the system complexity of a system modeled within a conceptual approach may then be shaped as shown in fig. 4.2, where it is compared to the natural evolution of a model starting with equal initial complexity but using a less conceptual framework.

In fig. 4.2 the natural system lifetime is determined by an upper bound for the complexity. Upper bounds constraining the system complexity do exist for almost every kind of system that arises 'naturally'. The reason is, that a system usually must provide particular functionality. For example, a system must typically react on external signals in acceptable time, which gets more and more difficult the larger its internal complexity becomes. Simulations of biological systems based on binary network models, show that the shape of a typical fit-

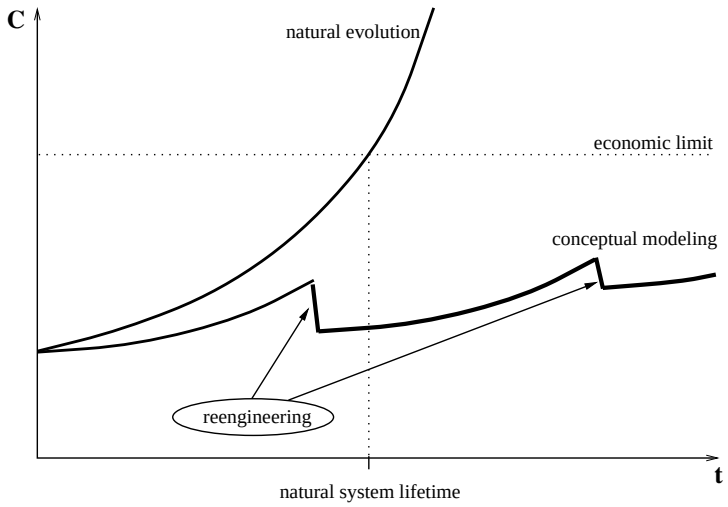


Figure 4.2: Increasing of system lifetime by reduction of complexity with the help of conceptual modeling characterized by a smaller modeling parameter and reengineering processes.

ness landscape becomes more and more fissured the higher the number of interconnections in the network becomes (epistatic coupling). The state of a biological system would then become suboptimal very easy, leading thereby to high instabilities [65]. In general, complex systems may evolve autonomously into a state called self-organized criticality, where small violations may produce generically cascades of changes, making thereby the system very hard to handle [5].

Although the reaction time is a criterion for information systems too, most of them are restricted to a lower level by economic criteria, as shown in fig. 4.2.

As mentioned above, one of the leading ideas of todays software engineering approaches is the so called object-orientation. Object-oriented approaches possess many advantages compared to the more conventional procedural approaches, but presuppose also much more structure at the beginning of a project. However, they are well-suited for more conceptual and theoretical considerations. In the next section we will state the basic ideas of such approaches and the existing methods to handle these ideas efficiently.

4.2 OBJECT-ORIENTED ANALYSIS

Object-orientation is a technique for developing models of general systems as information systems. Although information could mean nearly everything, object-orientation is of special interest when applied to systems describable by dynamical information entities called objects, that may be connected among each other by relationships that enable message flows.

In the following the main ideas of object-orientation will be explained. It is important to note, that the introduction of its main concepts as presented in the following is in agreement with object-oriented analysis, not necessarily with object-oriented design and programming. Object-oriented analysis is used to investigate complex systems with respect to the concepts of object-orientation, whereas object-oriented design tries to implement models as close as possible to these concepts. Due to technological reasons it is not always possible to do this in exact analogy.

One major advantage of object-oriented approaches when it comes to modeling, is the reduction of the semantical gap between model and modeled system, which arises due to some inherent properties of the specific modeling scheme. For instance, the separation of data and functions that mainly causes such a gap in procedural software approaches is rarely found also in the original system that is to be modeled. The reunion of data and functions into objects is a major topic of object-orientation, which may be characterized in agreement with the literature [66, 67, 68, 69, 70] by a few fundamental concepts, namely the concepts of *objects, types, classification, generalization / specialization* and *polymorphism*. In implementation considerations types are usually called *classes*, and classification is realized by so called *instantiation*. The generalization/specialization relations are implemented by a mechanism called *inheritance*. In the following both terminologies will be used simultaneously.

Consequently, object-oriented models consist of a collection of objects, each of them carrying an identity and representing a specific element of the modeled system. Every object is defined by a set of variables (*attributes*) their values describing the state of the element and by a set of behavioral rules specifying the operations that can be performed by the object and thereby the dynamics that governs its evolution. Objects can be mutually connected by relationships enabling for example the messaging of information from one object to another. Similar objects are classified into classes describing their common structure. Proper usage of object-orientation demands objects to arise exclusively as instances of existing classes, whereas not every class must possess direct instances. New classes can be achieved from existing ones by specialization which is usually realized by extending their properties via inheritance procedures. Polymorphism enables objects to communicate with each other without knowing the exact classes they belong to and provides thereby the ability to define operations that can act upon several distinct classes.

In order to apply object-oriented ideas efficiently, several system development methods have been proposed, describing either the development of a prototype or a global view of the entire system life cycle. After explaining briefly the basic object-oriented ideas, some remarks about the different existing methods are stated.

Object-orientation is considered as an anthropomorphic approach, motivated by the reduction of the semantical gap and the various analogies to human cognition.

Abstract Data Types

A well known concept in advanced procedural software engineering to overcome the somewhat artificial separation of data and functions, is the concept of *abstract data typing*. Abstract data types are abstractions of elements of the modeled system defined in terms of their internal information and of what actions they perform (and not of how they do it).

> An **Abstract Data Type** (ADT) is a collection of data and a set of allowed operations that are used to define and manipulate the data.

In this sense, ADTs generalize the usual data types like integers, reals or strings and the operations that apply on them, to more complex units. In addition, abstract data typing supports another advanced principle called *information hiding*[2], which means that system parts and in particular ADTs must be able to hide parts of their internal structure from their surroundings, showing only the operations that can be activated by external stimulations.

Abstract data typing and information hiding causes the system to possess much more conceptual structure since it separates clearly the pure modeling aspects from implementation issues. System modules are defined according to the activities that relate them to their environment and not to the specific way activities are implemented internally. Considered as a structuring principle, this is often called *modularity* and enables comfortable modifications of the system. Modularity is an essential idea in object-orientation, realized by its defining concepts.

Objects

In object-oriented models the most important dynamical role is played by so called objects. They can be viewed as run-time implementations of abstract data types, each of them additionally equipped with an identity and a state. Objects represent all identifiable agents of the modeled system. They possess a set of attributes and a collection of operations enabling them to behave in a specific way, once stimulated by an internal or external signal. The state of an object is the collection of actual values its attributes have and remembers the effect of the object's operations.

Once objects are defined, relations between them will be required. They represent the possibility of objects to know about and/or communicate with each other. For example, an operation of an object can include the transmission of a signal to another object activating an operation in the receiving one. Depending on their functionality, relations are sometimes classified into static ones and dynamical ones. Static relations represent the fact that a given object knows about the existence of another whereas dynamical relations are used for active information transfer. However, it is important to note, that in principle relations may connect more than two objects, that is, **multiple relations** are allowed. The exact analytical definition of relations will be given in the next section.

[2] Called *encapsulation* in object-oriented programming.

> An **object** is an element of the model having an identity and encapsulating possible states and behavior. Its state is represented by attributes and relationships, its behavior is represented by operations.

With relations another important item of object-orientation is supported, namely the composition of objects. This means, that it is possible to create more complex objects out of existing ones. The idea is to define a collection of objects to be a single new object, if there exists specific relations between them. A composite object then behaves like an ordinary object, when considered from its environment. Depending on their semantics two different kinds of whole-part relations are considered for this purpose, namely strong ownership relations and weak ownership relations. A **strong ownership relation** from an object a to an object b means both, that b is a part of a and that the existence of b depends on the existence of a. In contrast, **weak ownership** means that the part-object b may exist independent of the existence of a.

> A **composite object** is an object which possesses strong ownership relationships to a collection of other objects called its parts.

According to strong ownership this means, that no part-object can exist when the composite object is destroyed. One can state more general:

> An **aggregate** is the union of a collection of objects which are related by a multiple relationship.

Since multiple relationships cannot be easily represented, a new object possessing weak ownership relations to all objects of the collection is defined. This object *represents* the aggregate, but it is not the aggregate itself.

Types (Classes)

Usually many objects in a model will have similar information structure and behavior. Participating in the model they are distinguished only by their identities and their encapsulated actual states. To classify collections of objects with similar internal structure, types are introduced. They build a description of the common characteristics a collection of similar objects possess.

> A **type** is an idea or notion that applies to a collection of objects, while describing their common properties.

In this sense types serve for the semantical classification of the objects used in a model, and thereby the representation of the logical structure of the modeled system. For implementation issues, types are realized by classes.

> A **class** represents a template for a collection of objects describing how these objects are structured internally. Objects of the same class share the same attributes, operations, relationships. From a logical point of view they share the same semantics.

Classes are implementations of types. In the following they will be used often synonymously, since both represent the union of objects based on common properties. As mentioned before, abstract data types are defined as a set of operations describing a specific behavior. Since classes carry in addition to these properties the common information structure of the objects (attributes and relations) they describe, they can be considered as logical implementations of abstract data types.

The relations defined between objects may be lifted to relationships between the corresponding types. For example, the strong and weak ownership relations described above naturally extend to types with the same semantics, that is, representing composite and aggregate types and their strong or weak whole-part relationships to their constituents. Whole-part relations enable types to *delegate* messages to operations defined in their parts. However, it is important to note that the relations between classes may be multivalued, arising from relations between objects connecting a specific object of one type with several objects of another type.

Classification (Instantiation)

The *classification* of objects in object-orientation is represented by a relationship between the object and its classifying type.

> **Classification** is a relation that connects an object and a type that describes the information structure and behavior of the object.

Since in modeling a system, it is natural to start with the definition of the types that will be used, not only the union of common structure of existing objects must be enabled, but also the creation of new objects with exactly the internal structure contained in the classes. The mechanism by which objects are created from classes in object-oriented implementations is called **instantiation**. Objects arising in this way are called instances of the corresponding classes.

> An **instance** is an object. It is created from a class by instantiation. The class describes the entire structure of the instance except its identity, while the actual state of the instance is defined by the operations performed on the instance.

More than just classifying existing objects into classes, the instantiation concept of object-orientation demands that every object is actually a (direct) instance of exactly one class, that means that all objects in the model are created from corresponding classes. As mentioned before, in modeling a system conceptually, classes are defined at an earlier stage, so this is a natural assumption. However, since not every class must possess direct instances, classes which allow the instantiation of at least one object are called **concrete classes**. Non-concrete classes, called abstract, will be discussed later.

Applying object-orientation at this level, one may consider classes themselves as objects in the model, constructed according to more general templates called **metaclasses**. A model

describing the classes of another model is called a **metamodel**. Treating classes as objects, allows their participation in the dynamics of the model. Objects may communicate with classes, the operations activated in the latter influence every instance of that class.

Metaclasses are generically supported only in a few object-oriented programming languages (OOPLs) like *Smalltalk* or *Objective-C*, but they play an essential role in nearly all conceptual modeling methods. We will concentrate on metamodels in the next chapter.

Generalization/Specialization (Inheritance)

As the model gets larger by adding new types, some types will share common characteristics with other ones. Especially, a type may include the whole structure of another, adding further operations and therefore specializing the former one. To avoid this kind of redundancies, and thereby to enable a logical reduction of the model, another concept called generalization and specialization, is introduced in object-orientation. Generalization and specialization allow the representation of the logical structure the modeled system possess.

> **Generalization/Specialization** is a bidirectional relation that denotes the complete inclusion or encompassing of a type by another type. Generalization denotes the direction towards the encompassing more general type called the **supertype**. Specialization is the direction towards the included type, called the **subtype**.

In object-oriented implementation these relations are realized by *inheritance*, that is by transmitting the information of a class to its heirs. With the inheritance mechanism every class contains explicitly only the information that is specific for its individual behavior, while information common to other classes too is shifted into the information carried by some common ancestors.

> A class B **inherits** a class A if both the operations and the information structure described in class A is present in B. In this pairing the class A is called the (direct) **superclass** and B a (direct) **subclass** of A.

Inheritance is a realization of the generalization/specialization relations. Like them, inheritance is represented in object-oriented models by specific relations between classes. Nevertheless, inheritance is only one way to implement generalization.

The generalization relations yield a hierarchy of the used types. However, it is a highly non-trivial task to structure the requested types in a suitable way. Especially when new types are added, there are several possibilities to do this: one may find an ancestor in the inheritance hierarchy, or add the type independently. Both cases do not change the existing structure. But one may also try to restructure the hierarchy, so that a suitable ancestor results or redefine directly the characteristics of a given type. In these cases the logical structure of the model may change in an essential way. Another alternative, namely to change directly the characteristics in a type is called **overriding**. It is a controversy whether overriding

should be used in object-orientation or not. Actually it often destroys the transitive property generalization possesses, and demands a very careful treating of defined and eventually modified operations.

A standard obstacle that may occur when overriding is used originates in the fact, that theoretically inheritance is allowed to be a multiple relationship of more than two classes. A class is called **multiple inherited** when it possesses more than one direct ancestors. This is of special interest when the characteristics defined in two or more independent classes should be integrated in a single class. Now consider the hierarchy of fig. 4.3.[3] When the operation op$A$ inherited from class A is modified by overriding in the classes B or C, it is not clear which one should be inherited by class D. Stated system theoretically, this leads to frustration in the system.

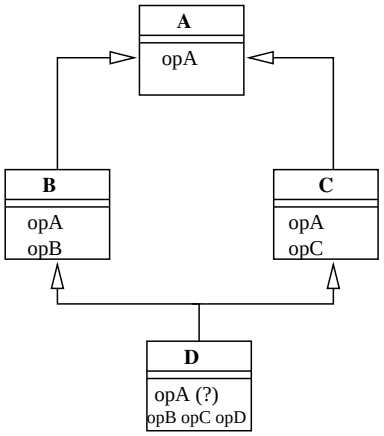


Figure 4.3: Obstruction due to simultaneously usage of multiple inheritance and overriding represented in UML notation. The arrows show inheritance relations, the source being the subclass, the target the superclass. The system is frustrated, if operation op$A$ is modified in classes B or C. The operation op$A$ is then ill-defined in class D.

Although from a purist's point of view one should not allow overriding at all, it is a very important feature of object-orientation when it comes to implementation. To avoid the lack described above most OOPLs allow overriding but do not support multiple inheritance. Nevertheless, since multiple inheritance has its advantages too, it is important that in general it can be replaced by delegation using aggregate classes, as shown in fig. 4.4. Being multiple inherited, the class D has access to the operations op$A$, op$B$ and op$C$, but the aggregation relation further enables conditions deciding which class should get an incoming signal and thereby overcoming the described problem.[4]

[3] To introduce a part of the *Unified Modeling Language* (UML) notation, examples illustrating the basic object-oriented concepts are presented within this language.

[4] Actually, in *C++* the constructor of a parent class is called first. This means that an instance of D constructs the operations op$A$, op$B$, op$A$, op$C$ and finally the operation op$D$. Multiple inherited operations must be

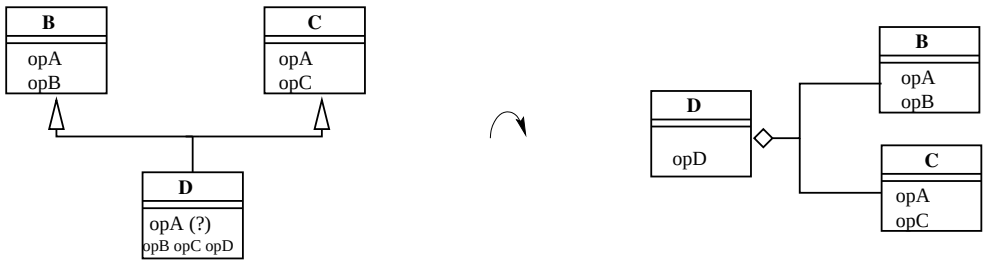


Figure 4.4: Multiple inheritance (left) is used to integrate the characteristics of two or more classes into a new one. Delegation (right) can be used alternatively and avoids obstructions due to overridden operations by stating conditions on which part-class will become the receiver of a specific signal. The diamond on the right hand side characterizes aggregation-relations in UML notation.

The generalization principle is a strong tool both, in analyzing and modeling a system, as well as in implementing it. In implementing a model, inheritance enables the reuse of code by encoding common characteristics in certain classes or by making available a class library from beginning. These classes contains the properties that are shared by a lot of possibly semantically unconnected classes, while making them available by inheritance. Classes that are introduced only as ancestors to inherit other classes, that is, which do not enable the instantiation of objects, are called **abstract classes**.

On the modeling side, generalization enables the representation of the logical structure of the modeled system by defining classes according to their semantics. To have precise semantics in the implementation region too, inheritance should support the **subtyping principle**, that is, the instances of a class should be replaceable by instances of all of its subclasses. Obviously, this is fulfilled in generalization by definition. Overriding should be used with respect to the modeled semantics to refine or delete the characteristics inherited from an ascendent as discussed in detail in [71]. As a general guideline it is reasonable to use overriding in an inheritance hierarchy preferentially when the considered class is abstract, since this is not in conflict with the subtyping principle by definition.

The object-oriented concepts introduced so far may be described as variants of a general principle called abstraction [72]. Here *abstraction* is the result of removing certain distinctions between things, so that a concept describing their commonalities is produced. In this sense, starting with objects, one arrives straightforward to types and classification. The generalization and specialization concepts are then considered as abstractions of types producing new, more general types.

overridden to avoid a *Compile-time-error*. (J. Würthner, private communication)

Polymorphism

Another feature supporting information hiding and proper usage of inheritance is polymorphism. Although it seems to be a rather technical detail, of interest only when it comes to implementation, this last characterizing property manifests another similarity to real systems. As known from human communication, if information is propagated from a transmitter to a receiver, it is the receiver who interprets the message. The transmitter carries at most information about what objects are able to receive the signal, but not about the concrete mechanisms activated in the receivers.[5]

> **Polymorphism** means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class.

Polymorphism allows models to be highly flexible. For example, adding another object to the model does not causes changes in the objects sending messages to it. From a modeler's point of view, it is not important how the operations defined in the classes are implemented.

Object-Oriented Programming Languages (OOPLs)

Object-oriented programming languages had been developed since the early 1970's. Although they differ in their syntax, most of them carry the ideas of classes and objects as their abstract data types and the concept of inheritance. Information hiding is often called *encapsulation* in OOPLs. Some OOPLs are (together with the year they appear):

> *Simula (1970), Smalltalk-76 (1976),*
> *Objective-C (1980), ObjectPascal (1985), C++ (1985), Eiffel (1988),*
> *ObjectCobol (1991), Ada 9 (1991), Java (1996)*

However, there are fundamental differences between the various OOPLs related to the enabling of multiple inheritance and also to which degree classes should be regarded themselves as objects. Higher *Smalltalk* versions are considered of the most proper OOPLs not allowing multiple inheritance and considering every class as an instance of a corresponding metaclass. At least until now the *C++* programming language is probably the most used OOPL. In *C++* multiple inheritance is allowed and metaclasses are not defined in an explicite way.

Object-orientation in Physics

There have been a plethora of efforts to use object-oriented ideas for modeling issues in various fields of science. Particularly worthwhile is the *GIBBS* project, developed in the late eighties for object-oriented modeling in lattice gauge theory [73]. An implementation of multigrid methods in *C++* code is found in [74] (see [32] for a ToCS-based approach and for discussion).

[5] In *C++* this is enabled by using interfaces.

To use its ideas efficiently, object-oriented modeling methods were developed, guiding and describing how models should be created. Modeling methods include notation techniques to represent a model. The *Unified Modeling Language* (UML), which will be discussed in detail later on, is a modeling language offering a substantial notation for object-oriented models, and supporting many properties of model development methods. One important reason for the arising of the UML was the necessity for a unique and widely accepted notation for object-oriented models.

4.3 OBJECT-ORIENTED DEVELOPMENT PROCESSES

The goal of software development processes[6] is to structurize and guide the development of information systems. An comprehensive development process usually consists of [75]

1. *Model elements* as building blocks of the model (definition),
2. *Notation* for their visual rendering (representation),
3. A *process* describing the development steps,
4. More or less formalized *know-how*.

While there is wide and good agreement about what the crucial ideas of object-orientation should be, lots of different approaches to handle the process of system engineering within this paradigm were developed in the past, leading also to considerable discrepancies between them. Basically, there are some few fundamental ingredients characterizing more or less the various existing system engineering environments. Most of them simply state which aspect in the development chain is considered as leading and governing the whole engineering approach. One may for example start by investigating the way processes have to be implemented, or with the hierarchical structure the used classes should have. Another approach could rely first on the definition of the applications the system must enable, ignoring the way they have to be implemented. Some main ideas, characterizing the different approaches may be described as follows [76]:

- **Iterative:**
  The iterative development process is driven by the definition of permanently iterated mechanisms, like the refinement of existing classes. While developing the system, these activities are iterated over and over to refine the executable system in a prescribed order. One disadvantage is, that realized substantial impact is not always explicitly documented.

- **Incremental:**
  This approach is an extension of the iterative process. The succeeding development cycles are defined by precise incremental improvements that must be achieved by completing the cycle. Therefore, demanded improvement defines future activities.

[6] Development processes are often called methods. Since, the term 'method' is used for implementations of operations, too, we prefer the term 'process'.

- **Evolutionary:**
  Evolutionary prototyping is a variant of the incremental approach. Every cycle, beginning with the first, produces an executable version of the system. The problem such approaches may confront one with is, that disadvantages in early versions may only hardly be corrected by following development cycles.

- **Event-oriented:**
  In some sense, this approach carries over object-oriented ideas into the development process. The system has to include all possible actions, their exact specification is realized as a reaction on an actual event (demand). The system is at every time formally complete.

- **Architecture-centric:**
  In general there are three kinds of architecture that govern the structure of information systems, namely the logical, the component and the physical architecture. These architectures characterize the internal structure of the used objects and processes, the structure of the interaction of software parts and the structure of the run-time system (hardware). Architecture issues are of special interest, when existing material has to be included in the system.

- **Use-case-driven:**
  This approach focuses on the typical applications the system must enable (use-cases), that is, what the system is expected to do. Use-case-driven development processes warrant the external behavior of the system, independent of its internal structure. Disadvantages originate in the inability to recognize and support optimization of internal issues.

As one may expect, most disadvantages of the different methodologies may be reduced by combining some of them into new integrate approaches. For example, a use-case-driven approach is best combined with an architecture-centric approach, avoiding thereby the inabilities of pure use-case-driven methods. Actually, this is done in almost every method used presently.

While the first object-oriented programming languages had been developed in the early 70's, usable object-oriented methods were first developed and published almost twenty years later [66, 68, 67, 69, 70]. Some approaches are shown in fig. 4.5, where their mutual influence is also shown in relation to the year they appear.

The methods that directly influenced the development of the UML are the *Object Modeling Technique* (OMT) of J. Rumbaugh [68], the *Object-Oriented Analysis & Design* method (OOA& D) by G. Booch [67] and the *Object-Oriented Software Engineering* method OOSE of I. Jacobson [66]. Both, OMT and OOA& D start their development process by specifying the classes that will be used later on in the model. This is extended in OMT by so called *entity/relationship modeling*, namely the definition of the logical structure the used classes must have and of all possible states the system can be in. In contrast, the OOA& D-method

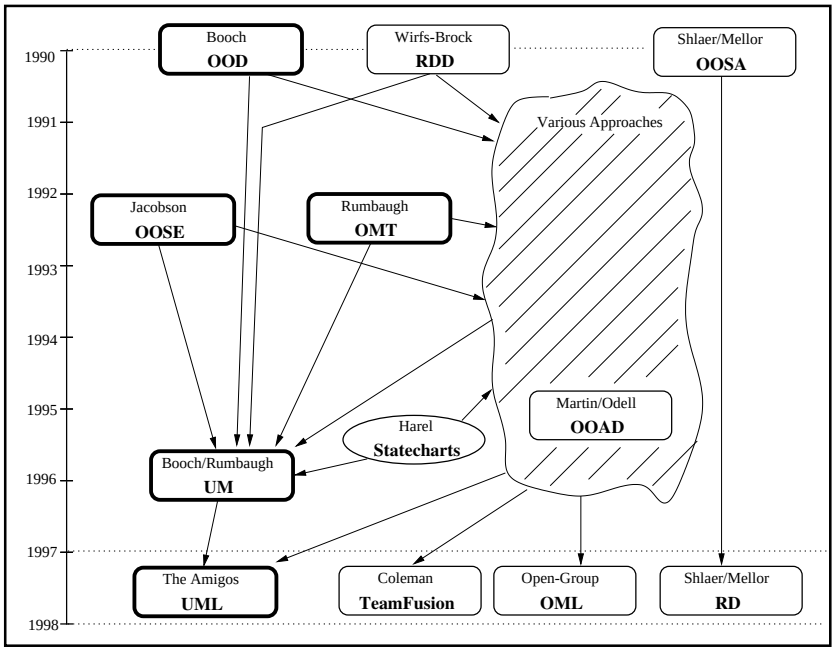


Figure 4.5: Some object-oriented methods and their mutual influences. The thick boxes show some of the approaches that directly influenced the development of the UML.

first equips the classes with semantics and then turns to the logical structure and its implementation, where the later is realized in an incremental approach.

In contrast to these approaches Jacobson's OOSE concentrates on the definition of typical applications of the system, starting thereby on a very coarse scale. The logical structure is introduced according to the needs of the specified applications and is used to define the demanded applications. The OOSE-method is the pioneer in use-case-driven approaches.

A widely accepted standard for all dynamical modeling issues concerning finite state transformations is the *statechart technique* of Harel [77] which is contained in the *Behavioral Elements Package* of the UML in its object-oriented version formulated by Harel and Gery at *i-Logix* [62]. Although the first version of the statechart technique was not developed for object-oriented systems, they have been used very early to describe the possible state transitions of objects in a visual fashion [78].

Other early development methods like *Object-Oriented Systems Analysis* (OOSA) by Shlaer and Mellor [79] or the *Responsibility-Driven Design* (RDD) by Wirfs-Brock et al. based on Class-Responsibility-Cards [69] can only be considered as preliminary, because they neglect several issues and concentrate only on specific tasks. The OOSA method for example does not describe inheritance and classification relations (logical architecture), where the

RDD method focuses strongly on the collaborations between classes and support architecture related topics only weak.

View architecture of the UML

It is of interest to ask as to what extend the characterizations given above may be used to describe the approaches that are supported by the UML. However, note, that the UML *is not a specific development method*, nor does it support a specific method exclusively.[7] Rather it is a general modeling language for the specification, visualization, construction and documentation of object-oriented models. Nevertheless, it is designed to support particularly development methods that are use-case-driven, architecture-centric, iterative and incremental. To enable this task, the notion of views is introduced in the UML. According to the UML documentation set [81], a view is defined as follows:

> A **view** is a projection of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.

As a consequence, a model is the integration of all of its views, whereas different views may intersect each other. The views of the UML are based on Kruchten's 4+1 view-approach [82]. This approach offers four complementary basic views, the *logical view*, the *process view*, the *implementation view*, and the *deployment view* and built upon them the *use-case view*. The logical view describes classes and their relations, while the process view describes the interactions of the run-time system. The implementation and deployment views concern with the architecture of the software modules and of the physical system respectively. The fifth view of Kruchten's approach, the use-case view, relies on the four others by gluing them into model-specific applications. Views are realized in the UML by accompanying diagrams (see appendix C). The next section investigates the formalization of the logical view of a model.

4.4 OBJECT-ORIENTATION AND TOCS

The attempt to formalize the whole dynamical content of an object-oriented model is a fascinating, but extremely challenging and sophisticated task. On the other hand, restricting the focus on the logical view of a model enables an almost straightforward formalization. In this section we concentrate therefore on the generalization/specialization hierarchy of an object-oriented model and on the different kinds of relations between classes and express this structure within the system theoretical framework. To avoid confusions note, that *the objects of an object-oriented model and the objects in a system are not semantically identical.*

The definition of classes given in the last section described them as entities representing

[7] A generic development process, that is based on the UML and unifies several accepted techniques is under development by the authors of the UML [80].

a set of objects together with a set of operations that apply on them. From a theorist's point of view, the set of operations can be considered as the defining common properties of the objects described by the class. Therefore, we formally define a class C to consist of a pair of sets

$$(\text{int}C, \text{ext}C),$$

the first term consisting of its characteristic properties (for example logical expressions), the second of all elements (objects) to which the logical expressions of the former set apply. The set of properties int$C$ is called the **intension** of the class C, the set of objects ext$C$ its **extension**.[8] To facilitate the following constructions we sometimes identify the class symbol C with its extension set ext$C$.

Objects and fundamental arrows

The system describing the classes and relations of an object-oriented model consists of classes and (object-oriented) objects as its ToCS-objects. To implement the whole generalization hierarchy and all further class relations, we consider a relation $\mathcal{R}$ between two classes X and Y as an ordinary set theoretical relation, that is, as a subset of the Cartesian product $\mathcal{R}(X,Y) \subset X \times Y$ consisting of all related pairs of objects.

It is important, that relations need not be single valued. To be able to treat multivalued relations properly we introduce functions $f_{\mathcal{P}}$ on the **power sets**

$$\mathcal{P}(X) := \{A \ : A \subset \text{ext}X\} \quad , \tag{4.4.1}$$

that will be associated to a given relation. For this, suppose the classes and relations of an object-oriented model are given. A relation $\mathcal{R}(X,Y)$ connecting objects of the classes X and Y canonically corresponds to the function

$$\begin{aligned} f : X &\rightarrow \mathcal{P}(Y) \\ x &\mapsto \left\{y \ : (x,y) \in \mathcal{R}(X,Y)\right\} \quad , \end{aligned}$$

assigning to each object x of X, the set of objects in Y to which it is related.[9]

For a composition of arrows to be well-defined, the function f must be lifted to a function $f_{\mathcal{P}}$ between the corresponding power sets

$$f_{\mathcal{P}} : \mathcal{P}(X) \rightarrow \mathcal{P}(Y) \quad ,$$

fulfilling the inductivity constraint

$$f_{\mathcal{P}}(A) = \bigcup_{a \in A} f(a) \quad . \tag{4.4.2}$$

[8] The definition of classes given here is motivated by the definition of *concepts* in the famous *Principia Mathematica* of Russel and Whitehead [83].

[9] The definition of relationships between data types as functions on power sets, was followed first in [84].

Conversely, starting with a function of the power sets $f_{\mathcal{P}} : \mathcal{P}(X) \to \mathcal{P}(Y)$, satisfying an analogous inductivity property

$$f_{\mathcal{P}}(A) = \bigcup_{a \in A} f_{\mathcal{P}}(\{a\}) \quad , \tag{4.4.3}$$

a function $f : X \to \mathcal{P}(Y)$ can be defined by $f(x) := f_{\mathcal{P}}(\{x\})$ leading thereby to a relation $\mathcal{R}(X, Y)$ connecting objects in the corresponding classes X and Y.

The fundamental arrows of the desired system are then given by the functions $f_{\mathcal{P}}$ on the power sets, corresponding to class relations and fulfilling the inductivity constraint (4.4.3).

To proceed in the definition of the system a composition law for the arrows must be defined. Since the arrows $f_{\mathcal{P}}$ are functions, we define the composition to be the ordinary composition of functions. To be able to interprete the composition of two arrows as a (composite) relation between the source and the target class, the composition must be compatible with the constraint (4.4.3). For this purpose, let $f_{\mathcal{P}} : \mathcal{P}(X) \to \mathcal{P}(Y)$ and $g_{\mathcal{P}} : \mathcal{P}(Y) \to \mathcal{P}(Z)$ be two functions arising from relations between X, Y and Z, satisfying the constraint respectively. For their composition holds

$$\begin{aligned} g_{\mathcal{P}} \circ f_{\mathcal{P}}(A) &= \bigcup_{b \in f_{\mathcal{P}}(A)} g_{\mathcal{P}}(\{b\}) \\ &= \bigcup_{a \in A} \bigcup_{b \in f_{\mathcal{P}}(\{a\})} g_{\mathcal{P}}(\{b\}) \\ &= \bigcup_{a \in A} g_{\mathcal{P}} \circ f_{\mathcal{P}}(\{a\}) \quad , \end{aligned} \tag{4.4.4}$$

which is the desired constraint and follows immediately from the inductive property of $g_{\mathcal{P}}$ and $f_{\mathcal{P}}$. Adding the identities to the set of arrows, we result with the system representing all classes and their relations.

Multiplicities

To be in close agreement with usual object-oriented definitions, some remarks on relations are necessary. First of all, relations in object-oriented models usually carry suggestive names (strings) like '*presentation*', or '*participates*', rather then symbols like 'f'. Obviously, this naming may be considered as a trivial isomorphism in the sense of ToCS. In a similar way, the pairs

$$\begin{aligned} & (f_{\mathcal{P}}, \alpha(f_{\mathcal{P}})) \\ \text{and} \qquad & (f_{\mathcal{P}}, \omega(f_{\mathcal{P}})) \end{aligned}$$

consisting of a relation-function and its source or target are called **relation ends** and are denoted by suggestive names called **roles** via string assignments.

Relation ends sometimes carry additional information. Suppose a relation $f_{\mathcal{P}}$ maps into the subset $\mathcal{P}_n(Y)$ of $\mathcal{P}(Y)$, consisting of all n-elementary subsets of Y, then the integer n corresponding to the relation end $(f_{\mathcal{P}}, \mathcal{P}(Y))$ is called the **multiplicity** of the relation end. In addition to the case of a specific multiplicity, it is also allowed to use multiplicity ranges, that is, to specify an interval or a general subset of $\mathbb{N}$ as possible multiplicities. In general, multiplicities may considered as maps that assign a multiplicity to each relation end.

Adjoint arrows

The system constructed above canonically extends to a complete system. The adjoint operation * for the arrows $f_{\mathcal{P}}$ is defined as follows

$$\begin{aligned} f_{\mathcal{P}}{}^* : \mathcal{P}(Y) &\to \mathcal{P}(X) \\ \{y\} &\mapsto \left\{x \in X \ : y \in f_{\mathcal{P}}(\{x\})\right\}, \end{aligned} \tag{4.4.5}$$

where to each object of Y the objects that are related to it by $f_{\mathcal{P}}$ are assigned. This function is extended to general subsets by the inductivity property. The operation (4.4.5) is a *-operation, the corresponding properties are easy to confirm

$$\begin{aligned} (f_{\mathcal{P}}{}^*)^*(\{x\}) &= \left\{y \in Y \ : x \in \{x : y \in f_{\mathcal{P}}(\{x\})\}\right\} \\ &= f_{\mathcal{P}}(\{x\}) \end{aligned} \tag{4.4.6}$$

and similarly for the adjoint of a composite arrow

$$\begin{aligned} \left(g_{\mathcal{P}} \circ f_{\mathcal{P}}\right)^*(\{z\}) &= \left\{x \in X \ : z \in g_{\mathcal{P}}\left(f_{\mathcal{P}}(\{x\})\right)\right\} \\ &= \bigcup_{\{y \,:z \in g_{\mathcal{P}}(\{y\})\}} \left\{x \in X \ : y \in f_{\mathcal{P}}(\{x\})\right\} \\ &= f_{\mathcal{P}}{}^*\left(\left\{y \in Y \ : z \in g_{\mathcal{P}}(\{y\})\right\}\right) \\ &= \left(f_{\mathcal{P}}{}^* \circ g_{\mathcal{P}}{}^*\right)(\{z\}) \ . \end{aligned} \tag{4.4.7}$$

The inverse of an arrow $f_{\mathcal{P}}{}^{-1}$ is defined as usual and exists if the arrow $f_{\mathcal{P}}$ admits a function on the sets X and Y which is invertible. The adjoint arrow $f_{\mathcal{P}}{}^*$ of an arrow cannot be considered as its inverse in the sense of power sets. Although it fulfils

$$f_{\mathcal{P}}{}^* \circ f_{\mathcal{P}} = \imath_{\mathcal{P}(X)} \quad ,$$

in general this equation does not hold in the opposite direction, that is,

$$f_{\mathcal{P}} \circ f_{\mathcal{P}}{}^* \neq \imath_{\mathcal{P}(Y)} \quad .$$

If the inverse arrow exists, it is identical to the adjoint arrow $f_{\mathcal{P}}{}^{-1} = f_{\mathcal{P}}{}^{*}$. Arrows that originate in specialization relations never possess inverses.

Generalization/Specialization

To complete the representation of the model in system theoretical notion, the generalization/specialization hierarchy has to be included. For brevity, we concentrate on the specialization relation. The generalization relations are the adjoint arrows to the specializations. The hierarchy is included by defining the specialization relations between classes as functions on the power sets in the sense described above.

However, for the purposes concerned here, the specialization relation is considered to be realized in form of inheritance restricted by the subtyping principle. Subtyping simply adds further logical expressions to the intension of a class, while restricting thereby its extension. For a class X inherited from the class Y, it holds int$Y \subset$int$X$ and ext$X \subset$ext$Y$. The inheritance relations can therefore be implemented as inclusion maps of the corresponding classes, that is, starting with the inclusions

$$j^{XY} : \text{ext}X \hookrightarrow \text{ext}Y$$

for classes X and Y, this lifts up to the power set functions

$$j_{\mathcal{P}}^{XY} : \mathcal{P}(X) \hookrightarrow \mathcal{P}(Y) \quad ,$$

where the constraint (4.4.3) trivially holds. Since the map $j_{\mathcal{P}}^{XY}$ is an inclusion of sets too, given a function $j_{\mathcal{P}} : \mathcal{P}(X) \to \mathcal{P}(Y)$ on the power sets, it is possible to check formally, whether or not it represents a specialization relation of classes.

Specialization relations are represented graphically in a system by arrows with a triangle tip. The transitivity property of inheritance relations is automatically warranted because of the composition law of the system.

To agree with common conventions, two classes X and Y will be distinguished due to the different names (or symbols like X, Y), even when their extensions and intensions are identical. In a system with generalization/specialization relations, the subtyping principle states, that no loops other than the identities exist.

Constraints on relations

As in ToCS, the specification of an object-oriented model by classes and relations may be restricted by specific conditions on the constituents called **constraints**. Although in principle they correspond to constraints in ToCS-sense, in object-oriented modeling they are used to restrict the way objects relate and how changes may occur. The former are called structural constraints, whereas the latter are called behavioral constraints. Once stated, con-

straints must be always fulfilled, as is the case in ToCS. There are some standard forms of constraints:

- *Cardinality constraints*
  Cardinality constraints restrict the possible multiplicities a relation may have. In particular, a *uniqueness constraint* restricts the multiplicity to be $n = 1$.

- *Order constraints*
  Order constraints require a multivalued relation to fulfil certain restrictions on the object sequencing. For example the `{ordered}` constraint states that the instances of a related class must be ordered. The directed acyclic graph constraint `{DAG}` or `{lattice}` ensures that an object cannot relate to itself. The `{tree}` constraint requires that a relation contains only one root object.

- *Mathematical constraints*
  Especially loop relations may be restricted by usual mathematical properties. Examples are the *reflexive* constraint, describing relations like equalities, or the *irreflexive* (objects do not relate to themselves). The *symmetric* constraint states that the relation and its adjoint possess the same semantics, whereas the *antisymmetric* constraint allows that only identical objects to be related. The *transitive* constraint allows the shifting of the semantics through composite relations.

- *Immutable constraints*
  Immutable constraints state that relations cannot be changed once stated.

- *Aggregation/composition constraints*
  Specify whole-part relations between classes or objects and weak or strong ownership.

The possible dynamics that may occur is restricted by behavioral constraints. They are stated usually in diagrammatic form like sequence diagrams or state machines and correspond in essence to the specification of the behavior that the model constituents are able to perform.

Objects are instances of a class, that is, they are the elements of the extension of a class. To illustrate the object-class relations in a system theoretical approach, too, the instantiation relation between an object x and its corresponding class X is represented by an arrow $X \to x$. This arrow is introduced here in opposite direction to the usual instantiation relation.[10] The reason for this choice is, that it enables the extension of the composition of arrows to the instance's level. The relations between classes define relations in the instance's level, connecting the specific instances that fulfil the relation of the corresponding classes (the concrete tuples of the relation). For documentation purposes it is notated in a system by a dashed arrow from a class to the specific object.

[10] To agree with the UML-notation, in the figures it is represented according to the UML-notation.

The construction of the system given above can be summarized as follows:

> **Result (Logical View):**
> *The logical view of an object-oriented model is a complete system whose objects are the classes of the model and whose arrows are functions on the power sets of classes which are associated to the class relations of the model and satisfy the inductivity property (4.4.3).*

Example

To illustrate the notation the following situation is shown. Given the classes X, Y and the class Z, which is inherited by X, an instance x of X and a relation $\mathcal{R}$ between X and Y yielding the arrows $f_{\mathcal{P}}$ and $f_{\mathcal{P}}{}^*$ that assign n objects of Y to each instance of X and m objects of X to each instance of Y, the system in fig. 4.6 represents this data in agreement with the UML notation following the construction described above. Relations are notated slightly different in UML, as explained in the next chapter.

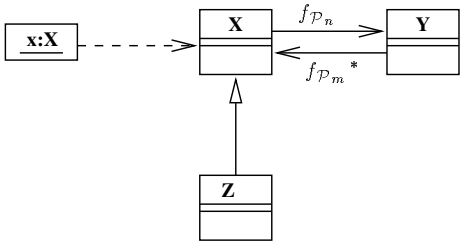


Figure 4.6: System theoretical description of the logical view of a simple object-oriented model. To agree with the UML notation and to identify generalization and instantiation relations visually, the corresponding arrows were notated dashed and with a triangle tip respectively. The arrow $f_{\mathcal{P}}$ and its adjoint represent the relation $\mathcal{R}$. The notation $\underline{x : X}$ symbolizes an instance x of X.

A simple example for the scenario described in fig. 4.6 is given in a triangular lattice. The class X is defined to be the set of triangles in a finite, periodic, triangular lattice, the class Y consists of all vertices in the lattice. The relation between both classes is given by the association of the corners to each triangle. This relation results in the functions

$$\begin{aligned} f_{\mathcal{P}} = f_{\mathcal{P}_3} &: \mathcal{P}(X) \to \mathcal{P}(Y) \\ f_{\mathcal{P}}{}^* = f_{\mathcal{P}_6}{}^* &: \mathcal{P}(Y) \to \mathcal{P}(X) \quad , \end{aligned}$$

that is, by the assignment of corners to a triangle and the adjoint assignment of all triangles that contain a specific vertex. The multiplicities are therefore given by $n = 3$ and $m = 6$,

with n and m as in fig. 4.6. A class Z may be given by any subset of triangles, whereas $\underline{x : X}$ is a particular vertex. The functions $f_{\mathcal{P}_3}$ and $f_{\mathcal{P}_6}{}^*$, and the corresponding relation ends may then be named anyway.

The object-oriented aggregation and composition of objects is in agreement with the concept of composite objects in ToCS. Strong and weak ownership relations have identical semantical meaning in both frameworks.

For multiple relationships to be defined, the notion of composite objects is used. This simulates the treating of multiple relationships by defining association classes in object-orientation and particularly in UML. Miming association classes, a composite object is introduced to represent the "cross section" of an multiple relation, transforming it into relations from the composite object to its constituents.

4.5 SUMMARY

In this chapter the object-orientation paradigm was introduced as an analysis concept for complex systems. Object-orientation treates complex systems as dynamical networks of objects and provides thereby premium candidates for systemic investigations. The role of software systems is explained and a foundation for object-oriented analysis is given. Some recent methodologies are described. Finally, the logical view of object-oriented models is given a natural system structure, enabling thereby the incorporation of object-oriented concepts in ToCS-based models.

Chapter 5

ANALYSIS OF THE UML METAMODEL

"In order to understand the description of the UML-semantics,
you must understand some UML-semantics."[81]

Object-orientation is very appealing in describing information systems like software as dynamical networks of interacting objects. It provides an efficient approach for modeling the most fundamental constituents and their direct relationships. Unfortunately, some more advanced features, like coupled sequences of interactions, or general collaborations of objects can not be appropriately treated. However, these aspects are essential steps to enable both, a precise specification of a model, and the abstract analysis and simulation of its properties. Architectural aspects are considered insufficiently, too.

The disadvantages of object-orientation do not only come from the failure of corresponding concepts, but are to some extend consequences of a missing formal backbone for the existing concepts. To overcome these difficulties, a somewhat more powerful framework, encompassing object-orientation, but providing more advanced and formally well-defined concepts is required. This approach should enable the specification and dynamical modeling of a system and should further provide precise mechanisms to analyze the models denoted within this framework. One of the most promising frameworks is the recent Unified Modeling Language approach. Based on the integration of several views, it encompasses the most common object-oriented modeling methods into a single framework.

The **Unified Modeling Language** (UML) is a modeling language for the specification, construction, documentation and visualization of models obeying the object-orientation paradigm, that is, for models in form of structured dynamical networks of objects. It is designed for modeling both, the pure structural and the behavioral aspects. For the specification and construction of models by well-defined modeling elements a precise semantical

framework called the **UML-metamodel** is provided. The UML-metamodel is a logical view and suppresses the technical and implementation aspects of a model. Although a logical view enables the accurate definition of underlying concepts, it has the disadvantage not to provide enough semantics for being directly implementable.

The visualization and documentation of UML-models is achieved by a collection of diagram techniques, called the **UML-notation**. The UML-notation document [85] specifies the correspondence of the graphical notation to the underlying semantics in a precise way. There exists a fast growing amount of literature introducing the UML. However, most of the literature concerns the UML-notation, trying to connect it to known object-oriented analysis and design methods [86, 75, 87, 76]. The metamodel itself, as well as related topics like its formalization, its extensibility or the comparison to other specification techniques is still on the stage of scientific and technical reports. Since the notation-technique of a modeling approach may be modified without changing the language content itself, this chapter will concentrate chiefly on the UML-metamodel. It provides a self-contained analysis of its concepts. To extend the comprehensibility of the UML-metamodel to a wider audience than the intended in the original document [81], the description presented in this chapter uses only natural language and the notation introduced so far.

The UML was developed in 1997 mainly by 'the three amigos' G.Booch, J.Rumbaugh and I.Jacobson at Rational Software Corporation as a unification of their corresponding system-development methods OOA&D, OMT and OOSE. It was proposed to the *Object Management Group* (OMG) as a universal standard for general purpose object-oriented modeling in September 1997 [88] and was accepted by the OMG on November 14, 1997. Due to its adopting and supporting by the largest IT-players, the UML is considered to become the leading modeling language in the nearest future, as can be observed right now in the field of system-engineering. However, developers and users of the UML claim its usage also in many non-scientific areas [89], for example in business process modeling of sales processes or development processes [90, 91].

The following sections describe the object-oriented metamodeling approach as necessary for the analysis of the UML-metamodel.

5.1 METAMODELING

This section concentrates on general object-oriented metamodeling. It addresses the following questions:

- Why should complex systems be metamodeled?
  What is the underlying structure metamodels should have?

- What is an object-oriented metamodel?
  How many metamodeling levels are required?
  Does a generic metamodel for object-oriented systems exist?

- How do metamodels enable precise and consistent notation?

General metamodeling

Every precise modeling approach should provide a collection of well-defined concepts and a corresponding notation. Of course, one may ask, how these concepts and notation can be defined within the model or, if necessary, in a broader context. A framework that defines the concepts, notion and usage of a model is called a metamodel for this model. It is a model for the information that can be expressed during modeling.

> A **metamodel** is a model that specifies other models. An object-oriented metamodel supports the fundamental object-oriented concepts and contains the types, whose instances are types for a corresponding model.

For object-oriented models, a metamodel should therefore provide a definition of classes (types), the classification of objects into classes, the generalization and specialization relationships between sub- and superclasses, and the concepts of relationships between classes. Obviously, for a specific model there may be various ways to define its elements and hence there will be in general more than one possible metamodel. Of course, metamodeling may be useful in non-object-oriented approaches, too. However, the missing classification concept makes its usage more difficult.

Metamodeling plays an increasingly dominating role in object-oriented system engineering. It is of particular importance in defining multi-view based modeling approaches like the UML. These approaches rely on the integration of several model specifications (views), each focusing some particular aspects of the modeled system, like the static structure, the dynamic behavior, or the functionality it provides. Describing the same system, these views are not independent. One has to assure their strict compatibility by stating how they are related. Metamodeling is a very promising approach to specify these issues and to define exactly how the different views are used to constitute the whole system.

Metamodeling aspects are of major interest not only from a theoretical point of view. In object-oriented development environments and CASE-tools, the ability to establish modifications on the used model foundations is closely related both, to a precise formulation and functional handling of its constructs. Usually, the modification of the collection of possible classes is realized implicitly by changes in the program code. Such a treatment frequently leads to semantical inconsistencies and fails in any way from being methodologically transparent.

Clarifying these aspects goes hand in hand with a deep and thorough understanding of the underlying model. The physical separation of model components for example by visual language for the logical hierarchy of classes and program encoding of their precise definitions, corresponds to a division of the model into several, almost unrelated sectors. As a consequence, one confronts inabilities to handle the whole model in a consistent and

uniform fashion. To avoid these ambiguities, a somewhat different logical framework for the underlying architecture is required. It should provide the modeling foundations, as well as the ability to extend them if necessary. The consistency of the used modeling elements should be stated explicitly.

The four-level metamodeling approach [70]

The *classification* concept of object-orientation introduces a natural separation of the model into two levels containing either, the classes, or their instances. Taking into account the metamodel level, too, where for example the class concept is defined as an abstract entity, yields three separate levels for the model. In addition, the demand to enable also modifications in the metamodel level itself, leads to the introduction of a metamodel for the metamodel, often referred to as the **meta-metamodel**. For example, the meta-metamodel should provide the foundation for the definition of object-orientation on the metamodel level. Taken this structure into account, this results in a natural four level metamodel architecture which is illustrated schematically in fig. 5.1.

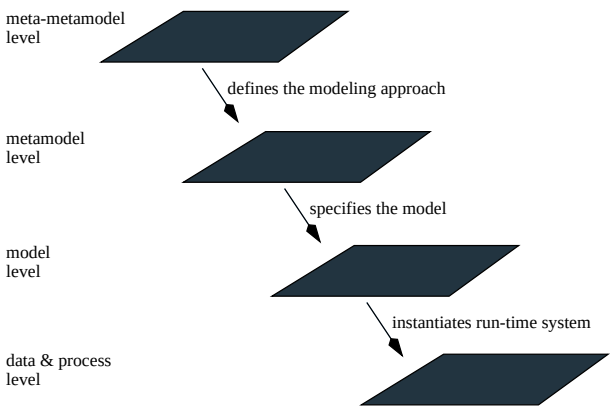


Figure 5.1: Four level architecture of object-oriented metamodeling.

In general, a metamodeling approach that clearly separates the modeling levels, is called **strict metamodeling**. In strict metamodeling, a template of an instance must be located on a higher modeling level than the instance itself [92]. The relation of an instance to its type is called *inter-level instantiation*. With respect to the logical view system introduced in the last section of chapter 4, strict metamodeling may be considered as a layered system, where the instantiation relations between different levels are given by unidirectional links from a higher modeling level M_i to the next lower one M_{i-1}. Each type then points towards its instances. The layers of the system are ordered, a higher label denotes a higher level of abstraction. To assure the consistency of the approach, the highest level should be a formal language that supports the definition of object-orientation. Of course, in a four level metamodeling, the layers are restricted to be M_1 to M_4.

Loose metamodeling

However, strict metamodeling may confront a variety of problems, which arise primarily due to the fact, that class-instances may serve as classes for further instances.[1] This means, that additional instances and thereby further modeling levels are required. In addition to the classification relations connecting the levels, one may require the existence of relationships between classes/objects and their corresponding instances. Such relationships lead to the mixing of the different modeling levels. This is a highly non-trivial aspect in most implementations and CASE-tools, as was already mentioned.

Object-oriented approaches encourage the use of a context-dependent designation of its model-elements as objects and classes. In addition, the ability to extend or specialize the model foundations is required in many cases, too. Together with the inability to relate elements of different levels, this motivates a somewhat more soft separation of the modeling levels. The most popular approach to overcome the discussed problems is **loose metamodeling**. In loose metamodeling an instance of a metaentity is allowed to occur on the same level as its template [92]. The corresponding instantiation relations are called *intra-level instantiation*, in contrast to the inter-level instantiation between different levels. Such architectures do not only evade the rigid separation of model elements, but enable relationships between arbitrary elements, too. The instantiation process is not restricted by a predefined number of modeling levels.

In any modeling level, an entity that allows intra-level instantiation of another entity in the same level, is called an **intensional entity**. Conversely, entities enabling inter-level instantiation of lower level entities, are called **extensional entities**. Obviously, this definition does not necessarily separate a model level into disjoint sectors. In [93], using a formal language (Object-Z) a precise definition of intensional and extensional entities is given. In particular, it is shown, that every intra-level instantiation relation from an intensional entity to an extensional entity in the metamodel level M_3, corresponds to a compatible inter-level instantiation from the model level M_2 to the data & processes level M_1. This is used in the following sections to structure the UML-metamodel, while respecting its semantics. For example, the intra-level instantiation of the *Object* metaclass in a metamodel, instantiated from the *Class* metaclass, is the abstraction of all ordinary class-object instantiations of the levels M_2 and M_1.

Like compilers are used to 'compile themselves' starting with a small core, in loose metamodeling one may try to explain parts of the metamodel by using its own elements. This means, that the model and the modeling approach should be described by the same con-

[1] Some object-oriented programming languages like *Smalltalk* and *Objective-C* support the notion of metaclass, whose instances, by definition, are classes, able to instantiate objects. A similar situation occurs, when powertypes are enabled, that is, types, whose instances are the subtypes of another type in the same level. Powertypes are frequently used for the classification of subtypes into species.

cepts. Obviously, this is only possible if all metamodel constructs are defined in consistent fashion. This assumes a precise notation, well-formedness rules, and axioms for its semantics. Miming the compiler paradigm, the meta-metamodel is then used to establish an elementary metamodel, called the metamodel core. Further metaentities are constructed by using both, the meta-metamodel as well as the metamodel core.

> A **metamodel core** is a metamodel that builds a conceptual foundation of a metamodel (of which it is a part). It is the basis for a complete specification of a metamodel.

The metamodel core should not be confused with the proper meta-metamodel, since not every metaentity is a core instance. Although loose metamodeling is not easy to handle formally [93], a metamodel based on a metamodel core may be thought as occurring in three steps. First, the metamodel core is instantiated in strict metamodeling from a meta-metamodel. Then, using both, the meta-metamodel, and the core, one proceeds in the construction of further metaentities. Finally, one integrates the core into the remaining metamodel entities, producing a metamodel level containing intra-level instantiations. Fig.5.2 illustrates the resulting architecture.

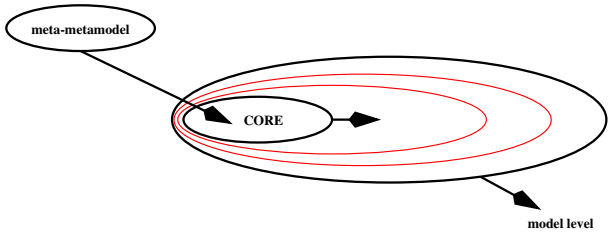


Figure 5.2: Metamodel with metamodel core in a loose metamodeling approach. Some instantiations of the metamodel level produce new metaentities.

Using a metamodel core called '*Core*', the UML-metamodel is defined in [81] as a loose instance of the *Meta-Object Facility* (MOF) meta-metamodel. The MOF is an attempt to provide a generic framework for metamodeling aspects. It is standardized by the *Object Management Group* (OMG) and manifests the object-orientation paradigm. In addition it addresses the modification of metaentities, for example, the creation and retrieval of metaentities.[2] The correspondence of the MOF-constructs to the UML-metamodel elements is explained in detail in [94, 88].

Of course, in object-oriented metamodeling, the metamodel core is used to establish the most fundamental concepts of object-orientation. Examples are:

1. Classes may associate to other classes. There is no general restriction on the multiplicity of the association.

[2] Increasing the risk of self-reference-confusions, the MOF is specified by using UML-notation (!), the semi-formal OCL (used in the UML-metamodel, too!), natural language, and CORBA specifications.

2. Superclasses possess generalization/specialization-relationships to any number of subclasses and vice versa (when multiple inheritance is permitted).

3. Objects possess classification-relationships to at least one class.

However, these conditions may be stated in any notation. Using the UML-notation, they take the form shown in fig. 5.3.

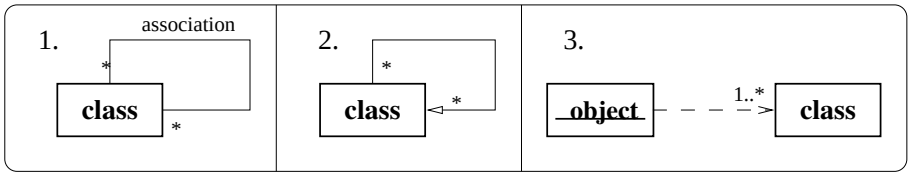


Figure 5.3: UML notation of generic core metamodel elements.

Summarizing, metamodeling is an elegant way to specify static and dynamic consistency constraints on the various modeling concepts. They are stated on metamodel level and hold for all derived model elements on lower levels. Moreover, in object-orientation the attempt to enable the extensibility of the modeling foundations requires a metamodel, too.

5.2 LANGUAGE SPECIFICATION

As described in the previous section, the specification of the UML relies on a four-level loose metamodel architecture. The metamodel level itself, defines the language that is used to specify a UML-model. A model must be expressed in terms of this language, that is, by instances of the metamodel elements. The UML-metamodel is a logical model, implementation and technical aspects of a modeled system are suppressed, although many concepts are introduced with the purpose to be implementable and to enable the construction of advanced CASE-tools. The elements of the metamodel are called **metaobjects**, with respect to their roles as instances of meta-metaclasses. In this sense, metaobjects may be metaclasses, metaattributes and metaassociations. Using these metaobjects, additional metalevels may be introduced as extensions to the defined elements. To provide a thorough and useful language, the metaobjects of the UML-metamodel must possess precise syntax and well-defined semantics. A commonly accepted methodology for the specification of a modeling language consists in specifying three basic dimensions of all metaobjects, namely their *abstract syntax*, their *static* as well as their *dynamic semantics*:

- *abstract syntax*
  The abstract syntax is an abstract description of all metaobjects that may be used to form a model of the respective language, and how they are built up eventually by other metaobjects.

- *static semantics*
  The static semantics of a metaobject defines its relationships to other metaobjects by

stating well-formedness conditions between the syntactical entities, for example the consistence of generalization relations by absence of circular inheritance.

- *dynamic semantics*
  The dynamic semantics of a well-formed syntactical metaobject is a description of its operational behavior, such as its reaction to stimuli or the effect of executing one of its operations.

The language used in the specification of the three dimensions of a metaobject may be a formal, a semi-formal, or a natural language. For the abstract syntax and also for the static semantics of a metaobject, it is convenient to make use of a formal, or at least of a semi-formal language, since for these aspects the well-formedness and consistency of the constructs is essential for the language to be well-defined. Due to the complexity of its content, the dynamic semantics specification is often not treated in a formal manner but in natural language. Obviously, using a natural or even semi-formal language for the specification of a metamodel dimension may lead to inconsistencies and requires therefore a lot of care. Furthermore, making use of more than one single language for the metamodel specification one has to assure the compatibility of the specifications by integration of the languages.

The original specification of the UML-metamodel in the UML-semantics document [81] uses parts of the UML-notation itself in metacircular fashion. The abstract syntax of the metamodel is described by stating UML class diagrams and is further explained in more detail in natural language. In this sense, one has to be familiar with the UML-notation to understand the UML-foundations. This self-reference property of the UML-metamodel requires great care to avoid incompleteness or even contradictions. The abstract syntax provides the generalization (specialization) hierarchy of the metamodel elements. Parts of the static semantics are also presented in UML-notation, namely the associations between metaclasses and their corresponding multiplicities. All further static semantics are given in form of invariants for the metaclass instances and of constraints for their attributes and associations. These expressions are referred to as *Well-Formedness Rules*.

For the definition of well-formedness rules an independent semi-formal language, the *Object Constraint Language* (OCL) is used, which is specified in an additional OMG-document [95]. The OCL is a mixture of natural language and programming code. Since the OCL is not explicitely used in this work, it will not be introduced in detail. Nevertheless, to enable the study of the original document a simple example is presented, namely the condition that circular inheritance is not allowed for any generalizable element in the model.
For this, assume the class X to be an instance of the *GeneralizableElement* metaclass, which is a supertype to all metaclasses that possess generalizable instances in the modeling level. Constituting a logical view, at least the abstract syntax of the UML-metamodel may be easily represented in form of a complete system as in the result of section 4.4 consisting of power sets $\mathcal{P}(X)$ and inclusion relations $j_{\mathcal{P}}^{XY}$ between them. The set of all supertypes

$J_g(X)$ of the class X is then given by

$$J_g(X) := \{Y \ : \exists \text{ path } p \ : \alpha(p) = X \wedge \omega(P) = Y\} \ ,$$

where the path p is a product of proper inclusion relations of the power sets of the corresponding classes. The required absence of circular inheritance is then given by the condition

$$X \notin J_g(X) \ , \tag{5.2.1}$$

which must hold for all classes $X \in$ ext(*GeneralizableElement*). In terms of ToCS, this simply means that the system representing the generalization hierarchy contains no loops (except for the identities). The condition (5.2.1) is given in OCL in very similar fashion. In OCL the expression '`self`' denotes the metaclass context to which the stated condition is referred to, for the case considered above this means that `self` is a class X which is an instance of *GeneralizableElement*. The set of all supertypes of an instance X is then given in terms of OCL by

```
self.allSupertypes
```

where the point '.' separates a set on its left hand side from a 'selector' on the right hand side that 'selects' a set of elements. The absence of circular inheritance is ensured by claiming that the class itself can never be included in this set

```
not    self.allSupertypes->includes(self)
```

where '**not**' means negation in OCL and the arrow '`->`' states, that the expression on its right hand side applies to the set on its left.

As illustrated in the example above, the well-formedness rules that specify the static semantics of the UML-metamodel can be represented as constraints for systems in the sense of ToCS. The dynamic semantics of the metaobjects are specified for collections of concrete classes in natural language, as already mentioned. From their definition, abstract classes do not possess any instances and thereby no dynamic semantics.

In order to increase the readability of the description, the UML-metamodel is introduced in the following by using logical view diagrams for the syntax of the metaobjects and natural language for additional description of the syntax. Natural language is used too for the definition of parts of their static semantics as well as parts of their dynamic semantics. Such a treatment is especially reasonable for the abstract syntax and the static semantics of the UML-metamodel, since these dimensions are defined in the semantics document [81] in formally satisfactory fashion. Due to their description as logical views, it is a straightforward task to translate them into ToCS as complete systems or at least as set-theoretical expressions. The translation into ToCS is performed as described in the corresponding section 4.4 of the previous chapter and as demonstrated for the static semantics example above. However, the dynamic semantics dimension of the UML-metamodel needs a thorough formalization when implementation of the dynamic semantics is required by advanced

CASE-tools. Attempts to formalize this aspect of the UML-metamodel have been proposed from the beginning of the UML development project and are a highly active area of research right now [96, 93, 97].

Similarly to the UML documentation set, for some metaclass *MetCla* the expression 'a *MetCla*' means 'an instance of the metaclass *MetCla*', as expected intuitively. Metaclasses of the metamodel are notated **boldface** when introduced in their definition and *italics* in the text.

5.3 LARGE SCALE STRUCTURE OF THE METAMODEL

After illustrating the architecture of the UML-metamodel and the structure of the language it specifies, this section introduces the metamodel elements with respect to this requirements. As mentioned in the previous sections, considering the UML-metamodel itself as a model, it respects the basic concepts of object-orientation. This enables one to define the metamodel in terms of classes, generalization hierarchies, relations and instantiation. In this sense, the elements of the metamodel are the types whose instances are used to specify a UML-model. The section explains the fundamental dichotomy of modeling concepts and representation issues and introduces the modularity concept that is used to structure both, the metamodel and derived models horizontally.

5.3.1 MODEL AND MODEL REPRESENTATION

Modeling a system essentially confirms the process of representing a given structure by using the elements and notion of another one. In this sense, a general modeling process usually separates three different sectors, namely the modeled system, the model itself, and the representation of the model in a graphical, textual, programming language or whatsoever notation. The three sectors are models themselves and may be the starting point for a further separation. In the UML-metamodel this separation is reflected in a very fundamental way by the structure of the top of its generalization hierarchy as seen in fig. 5.4.

The hierarchy in fig. 5.4 consists of three abstract metaclasses at its top, classifying the elementary constituents of the modeled system, the model and the model representation respectively. The *Element* metaclass is used to characterize the modeled system sector, whereas the *ModelElement* metaclass provides the most general structure for all modeling constructs. The *ViewElement* metaclass is used as a basis for all metaclasses needed in a representation of the model. In terms of the metamodel the definition of an *Element* is as follows:

> An *Element* is an atomic constituent of a model. **Element** is an abstract metaclass, being the supertype of all metaclasses in the UML metamodel.

In the metamodel the *Element* metaclass possesses two direct subtypes which are both abstract metaclasses. Usually a modeled system would be a physical entity that is to be

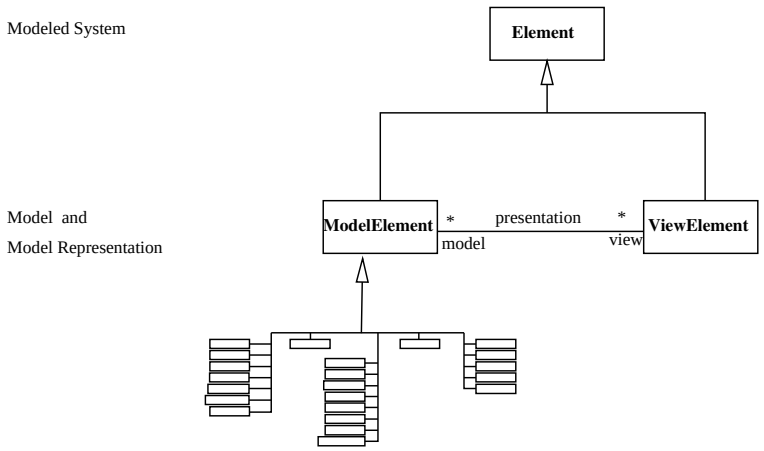


Figure 5.4: Separation of the elements of the modeled system and the model elements. Usually a model consists of its specification with the help of a modeling language and its representation by a notation compatible with the modeling concepts. In the UML this is realized by the above hierarchy where a *ModelElement* is every element usable in the model specification and a *ViewElement* is the supertype to all elements of a model representation.

modeled with the UML. The *ModelElement* metaclass is the head of the hierarchy of pure modeling metaclasses, whereas *ViewElement* the one of pure notational metaclasses. An analogous treatment of the representation is found in the system-theoretical implementation class library SyCL, although it is important to note, that it was developed independently (see Appendix A). In general, the out-placement of pure representational aspects in the described way is a widely used technique and reminds the structuring mechanisms based on design patterns [98].

> A *ModelElement* is an abstraction of an element of the modeled system. **ModelElement** is the top-most abstract metaclass of all modeling metaclasses. A *ModelElement* is a named element of a *Model*.

ModelElement is the supertype to all further metaclasses of the metamodel, except for the pure representation metaclasses. Subtypes of this metaclass provide the basis for the description of structure and dynamics of a complex system in a model. The definition of a model in terms of UML is given in the next section, where it is introduced as a structuring mechanism. The direct subtypes of *ModelElement* are shown in fig. 5.5 grouped semantically.

The further properties are specified in the next section concerning the *Core package* of the metamodel. Here one particular relation of *ModelElement* should be mentioned, namely the *presentation* relation connecting a set of *ModelElement*s to a set of *ViewElement*s, as shown in fig. 5.4. It specifies the details of the used representation of a model in graphical or textual form. A standard *presentation* relation for all modeling metaclasses is given in the notation specification [85]. The corresponding hierarchy of representation metaclasses

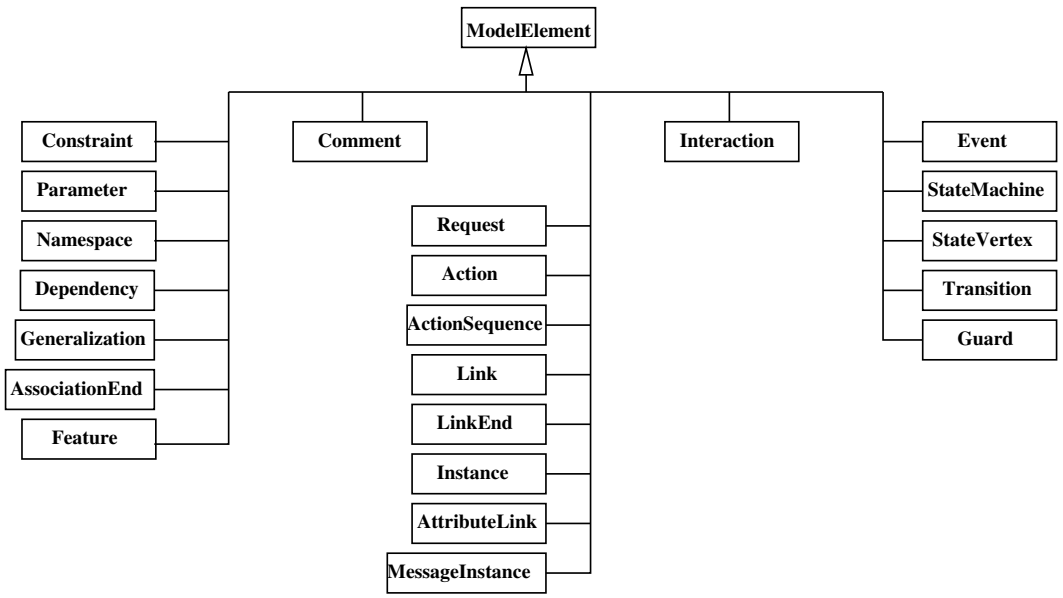


Figure 5.5: The direct subtypes of *ModelElement*. The left grouping shows the metaclasses needed to describe statical aspects. The middle grouping presents metaclasses describing the common behavior of model elements, whereas the right one shows metaclasses of the *State Machines* package, used to describe element behavior as finite-state transitions.

is inherited from the metaclass *ViewElement* which is a presentation of one or more model elements. In the metamodel:

> **ViewElement** is an abstract subtype of *Element* that presents a set of *ModelElement*s.

Subtypes of *ViewElement* correspond to subtypes of *ModelElement* via *presentation*. Since the presentation does not contain purely conceptual features of the modeling approach and depends moreover on whether a graphical or textual presentation tool is used, it is not further considered here. However, although the concrete realization of a representation depends on the used framework, it is important to note that the hierarchy of representation metaclasses must be defined in agreement with the *ModelElement* hierarchy in order to provide proper representations of the model. The UML-documentation provides a standard graphical representation of the metamodel elements.

5.3.2 HORIZONTAL STRUCTURING METHOD

In its standard form, the UML metamodel consists of approximately 90 metaclasses connected by 100 metaassociations and extended by nearly 50 generic stereotypes (variants of metaclasses). From a strictly logical point of view, one may expect to start their description with the definition of the most fundamental constructs (its core) and then to expand this core

by the construction of elements that rely on these fundamental metaobjects. Respecting the metamodel architecture this would end up with a collection of concepts on a certain level of abstraction.

Instead of proceeding this way, the large number of UML-metaobjects forces this section to provide a first overview, on the one side of how the elements of an abstraction level in a UML-model must be grouped into semantical blocks, and on the other side of how this structuring is applied to the UML-metamodel itself to yield a logical structure of the contained metaobjects. The loose or strict grouping of model elements of an abstraction level into different blocks according to pure semantical or logical criteria is called *horizontal structuring* of the model. The blocks arising this way need not to be independent of each other.

In general, formulating a language or a modeling approach with the help of a metamodeling procedure leads to considerable advantages, when compared to less structured treatments. In particular, it ensures, that each metaobject is defined as independent as possible from other metamodel constructs, by implementing the *modularity concept* via the definition of classes and relationships. This characteristic property of object-oriented metamodeling has proved to be of importance when it is necessary to extend or modify a model. In general, a technique that supports this feature is characterized as supporting the *localization principle* [97], which should not be confused with the notion of locality in the sense of ToCS. Actually there exists a plethora of methods to implement the localization principle on various scales in a given model. In this sense, the separation of model elements into horizontally structured blocks supports the localization principle on a large scale, if elements in different blocks are related as minimally as possible.

Horizontal structuring methods of the UML

Once completely formulated, the UML-metamodel may be used to define other models. Since metamodels can be considered as models too, they may be described using the UML, provided they underly the object-oriented paradigm. To some extend, this approach is followed in the UML-documentation set [81], where the metamodel is described using the UML notation [85]. In particular, the structure of the metamodel is described by using those constructs of the UML, that describe the packaging of the semantical parts of a model. To avoid such confusions, an informal overview of the structuring method is given.

The UML offers a specific horizontal structuring method defined in its own terms. This is done by providing the concepts of packages, models, and subsystems in the metamodel. These metaclasses are distinguished due to the structure of their content and have different semantics. The most general grouping mechanism is offered by the *Package* metaclass which simply provides a name-space for a plain grouping of elements and a loose relation to other *Package*s. It possesses two direct subtypes, the *Model* and the *Subsystem* metaclasses (fig. 5.6).

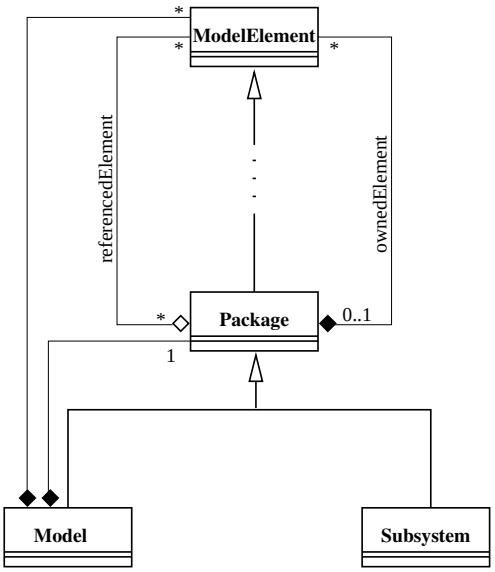


Figure 5.6: The *Package*, *Model*, and *Subsystem* metaclasses contained in the Model Management Package. The shown relations are instances of metaclasses introduced in the Core package.

In the logical hierarchy of the metamodel, the *Package* metaclass is defined as follows:

> **Package** is an abstract direct subclass of *GeneralizableElement*. A *Package* owns a collection of *ModelElement*s, and may refer to a set of *ModelElement*s in another *Package*.

The *GeneralizableElement* metaclass will be defined in the Core package introduced in the following section. It enables the generalization of a package, in the sense, that the specialization has the same access to referenced *ModelElement*s as has the parent *Package*. The corresponding visibility (private, protected or public) of its *ModelElement*s is defined in the original *Package*. The *Package* has strong ownership of its elements, removing the *Package* eliminates simultaneously its constituents. Different *Package*s may be related among each other by *Dependency* relations, which will be defined in the following section. Naively spoken, *Dependency* relations state, that modifications in a *Package* may influence the dependent *Package*. Of course, a *Package* contained in another one depends on its container. The generalization and the dependency relations are the only connections of *Package*s, assuring thereby the most possible independence of the various *Package*s. Both relations are transitive and specify the *ModelElement*s that are referred to by a *Package*. Other relationships are allowed only among *ModelElement*s contained in the same *Package*.

The *Package* metaclass may be used in various contexts and does not possess predefined meaning. It possesses two direct subtypes specializing this general grouping mechanism for specific requirements. The *Model* subtype has the purpose to unify the *ModelElement*s that describe a view of a modeled system at a specific abstraction level, for example its

logical view in form of classes and relations.

> **Model** is an abstract *Package* subclass. A *Model* contains *ModelElement*s in form of a *Package* hierarchy specifying a modeled system completely and in form of *Actor*s that describe the modeled systems environment.[3]

In this sense a *Model* is a complete self-contained description of the modeled system suppressing only those issues that are not relevant for the chosen point of view or that correspond to a different level of abstraction. It contains at least one *Package* (see fig. 5.6). Consequently this top-most *Package* of its contained hierarchy describes the boundary of a view of the modeled system. The environment of the modeled system may be contained in form of *Actor*s and associated *Interfaces*. The relations between the environment and the 'internal' elements of the model are also contained in the *Model*, that is, the *Model* may be considered as the transitive closure of all owned elements. Being a *Package* subtype, a *Model* may possess a generalized *Model*, whose elements are available in the specialization. Further relations between different *Model*s are used only for description purposes and do not influence the semantics of the contained *ModelElement*s.

The second subtype of the *Package* metaclass is the *Subsystem* metaclass. It is used as a grouping mechanism that additionally provides the possibility to define the behavior of its contents with the help of *Operation*s and *Interface*s. For this its contents separate into specification elements and realization elements, where the former together with the *Operation*s describe the behavior of the *Subsystem*s contents and the later how this behavior is realized on a lower level of abstraction by cooperating realization elements. Usually specification elements are *UseCase*s and *Operation*s, whereas the realization elements are *Class*es and *Subsystem*s. The relations between both sectors are given in form of *Collaboration*s. Using the provided *Operation*s a *Subsystem* may possess a number of *Interface*s that may be shared with other *Subsystem*s.

> **Subsystem** is a subtype of the *Package* metaclass (and of the *Classifier* metaclass defined in the Core package). It contains *ModelElement*s and a set of *Operation*s.

A *Subsystem* may be instantiable, in the sense, that all of its contents are instantiable. Even if this is the case, the specified behavior of the *Subsystem* refers to its realization elements. This is true also for the connections of the *Subsystem*. The *Collaboration*s then show how the instance of the realization elements play together for the realization of the behavior specified by the specification elements. The generalization and *Dependency* relations are used for *Subsystem* in the same way as for general *Package*s.

The *Subsystem* metaclass provides a very useful concept since it enables the description of the dynamic behavior of a part of the whole model without specifying exactly the realization of this behavior. Therefore it is of use when the model specification starts on a

[3] The *Actor* metaclass is defined in the context of *UseCase*s later on.

coarse scale and is developed in a top-down fashion.

The horizontal structure of the UML-metamodel

After introducing the horizontal structuring mechanisms provided in the UML-metamodel it is natural to apply these mechanisms to structure the metamodel itself. This is possible consistently, since one may think of an a posteriori structuring. The grouping of the metaobjects is performed by using the *Package* metaclass. The package structure of the metamodel and its dependencies is shown in fig. 5.7.

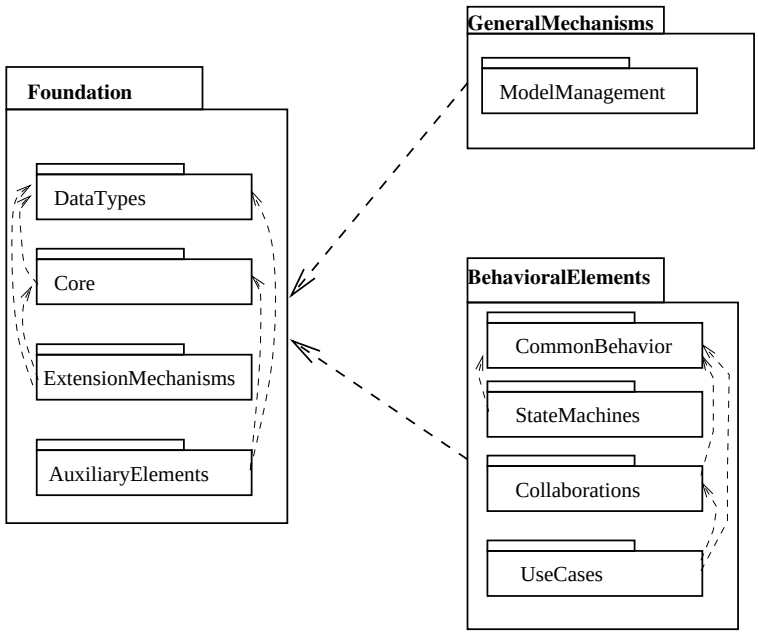


Figure 5.7: Horizontal structure of the UML metamodel. Three semantical blocks, focusing either the static, the dynamical or the structuring aspects exist. The folders represent *Package*s in agreement with the UML notation. Dashed arrows represent *Dependency* relations.

The whole metamodel separates semantically into three principal packages, the *Foundation*, the *Behavioral Elements* and the *General Mechanisms* package. The Foundation package contains the *Core*, the *Auxiliary Elements* and the *Extension Mechanisms* packages. The Core package defines the basic concepts of an elementary core metamodel providing the metatypes of which the metaobjects contained in other packages are specializations of. The Auxiliary Elements package extends the Core by certain more advanced constructs, whereas the Extension Mechanisms package specifies how the semantics of model elements may be extended and fitted into their context. The Behavioral Elements package describes the metastructure of behavioral modeling in UML: the *Common Behavior* package contains the core behavioral concepts, the *Collaborations* package enables the task-specific interaction

of objects, whereas the *State Machines* package defines the internal dynamics of objects as finite-state transition automata. The *Use Case* package specifies the interaction of a model system with its surrounding actors. Finally the General Mechanisms package contains so far only the *Model Management* package, which defines the way UML-models should be structured, that is the horizontal structuring mechanisms defined above. These packages are not independent of each other, their dependencies are shown in fig. 5.7.

5.3.3 Common Mechanisms in the Metamodel

The large number of metaobjects of the UML-metamodel may be horizontally structured in a somewhat implicit way. This additional structure arises mainly by the fact, that the metamodel reflects many properties of an object-oriented model. There are three distinguished common mechanisms:

- **Common Dichotomies**
  Two basic kinds of division mechanisms realize some generic dualities in the metamodel, namely type/instance dichotomy and the interface/implementation dichotomy.

  The *type/instance* dichotomy describes the fact, that most metaobjects arise in pairs of related metaobjects, the former being an abstraction, the later a concrete manifestation of the former. In this sense there is a Class/Object pair, an Association/Link pair, a DataType/DataValue pair, a UseCase/UseCaseInstance pair, and so on. The type/instance dichotomy corresponds to the definition of intensional and extensional entities in the first section. This dichotomy is important when modifying the metamodel: intensional entities change the modeling approach, whereas extensional entities change the model. This dichotomy is used in the following to relate Core and Common Behavior elements.

  The *interface/implementation* dichotomy describes pairs of metaobjects where the former denotes an abstract requirement and the later a concrete realization of this requirement that specifies an implementation compatible with the required interface semantics. Typical examples for this are the UseCase/Collaboration pair, or the Operation/Method pair.

- **Extension Mechanisms**
  As will be described later on, the UML provides standard mechanisms to extend existing models. These mechanisms are used extensively to extend the metamodel itself. There are three ways of doing this: by definition of new types (Stereotypes), by extending properties of existing types (TaggedValues), and by the extension of the semantics existing types posses.

  Many metaobjects arise as extensions of basic metaobjects in the sense of the above extension mechanisms. The metamodel may therefore be structured with respect to its basic, not extended types.

- **Model-Representation Specification**
  As already discussed, the *presentation* relation connecting the model to a representation is not necessary 1-1. In particular, this means, that more than one model element may be used to define a representation element. In this sense an afterward structuring of the metamodel may arise, once a representation is specified, showing the grouping of model elements that specify a representation element. In addition, collections of such groupings may arise, when representation elements are related in the notation.

The effect of the above common mechanisms on the metamodel are visualized naively in fig. 5.8

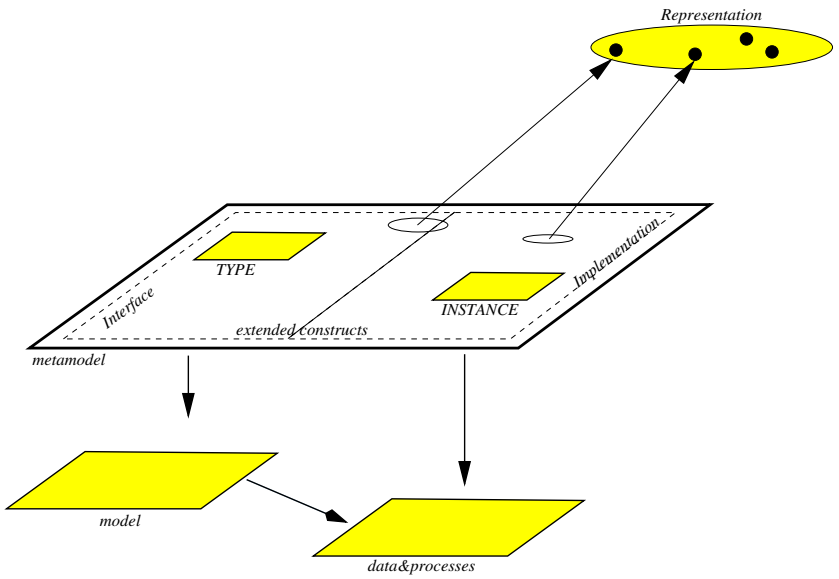


Figure 5.8: Horizontal structuring of the metamodel due to common mechanisms. The figure shows the non-extended blocks and the dichotomies that divide them. Intra-level instantiation in the metamodel corresponds to inter-level instantiation between model level and data & processes level. Representations may group the metaobjects according to the specification of their representation elements.

5.4 MODELING STRUCTURAL PROPERTIES

The static aspects of the UML-metamodel are described in the *Foundation* package. This package consist itself of four packages, the *Core* package, the *Extension Mechanisms* package, the *Auxiliary Elements* package and the *Data Types* package.

The Data Types Package

The Data Types package of the UML-metamodel contains the basic data types that are used in the definition of the UML and plays a somewhat exceptional role in the UML-metamodel. Their first use in the metamodel is to provide the types whose instances basic class-attributes should be. Due to their fundamental role, they are represented directly by strings in UML-diagrams. Metaobjects in the Data Types package are subclasses of the *DataType* metaclass defined in the Core package. Nevertheless, to avoid confusions, it is important to note, that the instances of the *DataType* metaclass defined in the *Core* package and the elements of the *Data Types* package are semantically distinguished. The instances of the *DataType* metaclass are the data types that will be used by a UML-user in a model, whereas the metaobjects in the *Data Types* package are the data types that are used for the definition of the metamodel itself.

Most metaobjects in the *Data Types* package separate according to two subclasses of the *DataType* metaclass, namely the *Primitive*, *Enumeration* metaclasses. These metaclasses in the package serve as superclasses for further metaclasses:

Primitive
Primitive is a specialization of *DataType* without any relevant substructure. Specializations of the *Primitive* metaclasses are **Integer** ($\mathbb{Z}$), **String** (stream of text), **Time** defining a value which represents an absolute or relative moment in time and space, and **Uninterpreted** being a blob, without UML-defined meaning.

Enumeration
Enumeration is a *DataType* specialization, consisting of a list of definable values called **EnumerationLiteral**s without substructure that are comparable for equality. Specializations of the *Enumeration* metaclass are for example the **Boolean** metaclass defining an enumeration with values {true, false}, the **AggregationKind** metaclass with values {none, shared, composite} denoting the character of an *Association*, the metaclass **ChangeableKind** denoting by the values {none, frozen, addOnly} how **AttributeLink** and **LinkEnd** metaclasses may be changed, the **OperationDirectionKind** metaclass denoting by the values {provide, require} if an *Operation* is required or provided by a *Classifier*, the **ScopeKind** metaclass defining an enumeration with values {classifier, instance} denoting if a stored value should be an instance of a *Classifier* or the *Classifier* itself and the metaclass **VisibilityKind** with values {public, protected, private} defining how the referred element is available from outside the enclosing name space.

Some further useful metaclasses of the Data Type package has to be mentioned. They provide types to classify attributes of *ModelElement* by their names, their composition by a number of parts or by obeying constraints given in form of evaluable expressions.

> **Structure** is a specialization metaclass of *DataType*, that has a fixed number of named parts. Further, **Geometry** is a metaclass denoting a position in space as a triple of floats, the **Multiplicity** metaclass defines a non-empty set of non-negative integers to characterize multiple associations, the **Expressions** metaclass defines a statement that evaluates to a set of instances when executed without changing its evaluation environment and the **Name** metaclass defines a token for naming instances of *ModelElement* and possesses a corresponding *String* representation.

Due to the primitive usage of the *Data Types package* in the metamodel and in the modeling approach, this package will not be further analyzed.

5.4.1 THE CORE

As described above, object-oriented metamodels are specified either, in strict, or in loose metamodeling. The UML-metamodel follows a loose metamodeling approach. The most fundamental metaobjects are contained in the *Core package*, which builds the metamodel core in the sense described in the first section of this chapter. Being a metamodel core, it is not the meta-metamodel of the UML, but almost every metaobject in further packages uses the Core package. The specific dependencies of the metamodel packages, illustrating the fundamental character of the Core are shown in fig. 5.7.

Instantiation

To enable generic object-oriented models, the metaobjects contained in a metamodel core should offer a substantial manifestation of the basic concepts of object-orientation. In the UML-metamodel, the Core package provides concepts for a proper realization of objects and types, generalization and classification. In particular, the classification relation is realized by instantiation. This means, that the system that is to be modeled is described by a collection of states, where every state is a complete specification of all objects, links and values. These entities arise directly as instances of a corresponding unique abstraction. The instantiation process produces objects, whose information content consists of exactly one consistent value for each attribute of the corresponding class. Similarly the information content of a link is a tuple of instances, that are related by the link in a way consistent with related classes in the corresponding association. Consequently, every object originates from a unique direct class, whereas every value belongs to a unique direct data type and every link to a direct association. In the reverse direction, run-time elements are called direct instances of a corresponding abstraction, in contrast to indirect instances when the specific abstraction is an ancestor of the direct element.

The Core package elements are recursively applied in the definition of more complex metaobjects.The generalization hierarchy is shown in fig. 5.9. In the following the main aspects of the abstract syntax and semantics of the metaclasses are described as close as possible to the hierarchy 5.9.

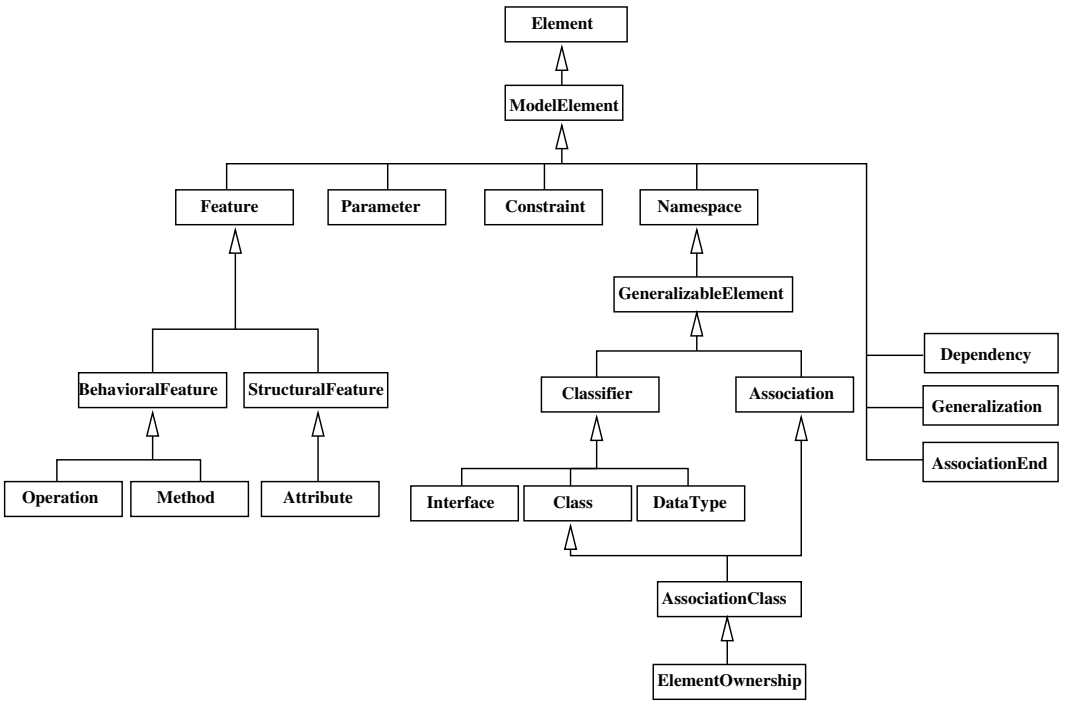


Figure 5.9: The metaclasses hierarchy of the Core package. Most of the metaclasses defined here serve for the definition of metaclasses in other packages.

To present a self-contained but compact description of the metamodel, the syntax and the semantics of the metaobjects are given in an integrated fashion.

Naming and constraining elements

The *Element* metaclass and parts of the *ModelElement* metaclass were described in previous sections on the modeling-representation dichotomy. The *Element* metaclass is not equipped with any further properties. However, *ModelElement*s do participate in a number of relations. In general, a *ModelElement* may participate in *Dependency* relations both, as dependent element (with role-name *requirement*), and as supplier (*provision*). *Constraint*s on a *ModelElement* must be true for the system to be valid. In addition, it possess a unique *name* attribute. For this, it is contained in a *Namespace* by an association which defines its visibility outside the containing *Namespace*.

The *Namespace* metaclass provides the possibility to define unique names for the *ModelElement*s contained in a subset of a model.

> **Namespace** is an abstract *ModelElement* subclass. A *Namespace* owns a set of *ModelElement*s specified by the *ownedElement* relation. Each *ModelElement* contained in the *Namespace* has a unique name and may be owned at most by one *Namespace*.

*ModelElement*s contained in a *Namespace* are connected by the *ownedElement* association. A *Namespace* should provide names for a unique meaning of a part of a model. However, some owned relations (*Association* and *Generalization*) may violate uniqueness of their naming, although the related elements may not.

The semantics of *ModelElement*s are not always adequate for specific situations. As mentioned in the section on the extensibility of the metamodel, one generic mechanism to extend the semantics of given elements in a required way, is to state semantic restrictions on the *ModelElement*s. Technically, in the UML this is realized by connecting the considered *ModelElement* with an expression that admits a boolean evaluation. These expressions are called *Constraint*s and their specification is possible in any language, provided that the condition is a well-defined boolean one. Any number of *Constraint*s may be related to at least one *ModelElement*. In the *Association*, the later has the role name *constrainedElement*.

> A **Constraint** is a *BooleanExpression* referred to a set of *ModelElement*s. *Constraint*s must always be true when evaluated in a well-defined model.

*BooleanExpression*s are specializations of *DataType*s, which evaluate to an instance of *Boolean*. The *BooleanExpression* is contained in the *Constraint* as the *body* attribute. A *Constraint* is not executable, rather it states conditions for the system to be valid. In the metamodel, some predefined *Constraint*s exist. They are used to extend the semantics of metaobjects. To avoid problems of self-reference, *Constraint*s are not allowed to be applied on themselves.

Full descriptors, generalization and generalizable elements

The creation of an instance relies on a complete specification of all attributes, associations and operations that are defined by a corresponding *Classifier*. The complete definition of an instance structure is called a full descriptor and is achieved via the generalization property of object-orientation by segments of information, that are combined (added together) for the instantiation using inheritance. In the UML, elements that may participate in an inheritance hierarchy are called generalizable elements. By inheritance, the instance of a generalizable element inherits the features of this element and all inheritable features of its ancestors. Inheritable features of a generalizable element are its attributes, its operations, its signal takers, its methods and its participation in associations. Further, constraints on model elements are inherited in any inheritance procedure. The inheritable features must be declared uniquely in a full descriptor. Methods may violate this property, since every method replaces ancestor-methods with the same signature. However, in an inheritance hierarchy two equivalent methods are not allowed to remain in an instantiation. Methods in

a full descriptor must correspond to operations. In this sense, the full descriptor consists of the segment descriptor of the generalizable element whose direct instance is considered and of the union of all segment descriptors of ancestors.

The generalization concept of object-orientation is realized in a UML-model by instances of the *Generalization* and the *GeneralizableElement* metaclasses of the metamodel.

> **GeneralizableElement** is an abstract subclass of *Namespace*. Every type whose instances may participate in a *Generalization* relation is a *GeneralizableElement* subtype.

To describe the position of a *GeneralizableElement* in an inheritance hierarchy it contains some boolean attributes, like *isLeaf* stating if no further descendents to this element exist in the hierarchy, or *isRoot* showing if the element has no ancestor. The *isAbstract* attribute states if the *GeneralizableElement* does not provide a full descriptor for instantiation, that is, if it is an abstract element. Uniqueness of the naming of elements requires the supertype of a *GeneralizableElement* to be contained in the same *Namespace*.

A *GeneralizableElement* may be related to the *Generalization* metaclass. This metaclass denotes a relation between *GeneralizableElement*s in a model, that shows, that an element contains all information like *Attribute*s, *Operation*s and participation in relations of another element, but may be equipped with further properties.

> **Generalization** is a direct *ModelElement* subclass. A *Generalization* refers to a set of *GeneralizableElement*s specifying a directed relation between them. It defines a taxonomic relation realizing the subtyping principle.

Subtyping means that *Instance*s of a *GeneralizableElement* may replace *Instance*s of a more general *GeneralizableElement*. According to their participation in a *Generalization* relation, the *GeneralizableElement*s are denoted by *supertype* or *subtype*, the corresponding relations are *generalization* and *specialization*, respectively. *Feature*s defined and *ModelElement*s contained in a *GeneralizableElement* are accessible also in the heirs of the parent *GeneralizableElement*. To group subtypes of a *GeneralizableElement*, an inheritance partition may be named, motivating thereby an attribute *discriminator* in the *GeneralizableElement* which denotes this partition. Partitions are useful to describe similar semantical properties of subtypes.

Classifiers and their features

The most abstract metaobject that unites structural and behavioral properties in an entity is *Classifier*.

> **Classifier** is a direct, abstract subclass of *GeneralizableElement*. A *Classifier* is used to declare a collection of *Feature*s (*Attribute*s, *Method*s, *Operation*s) with a unique name in its containing *Namespace*.

The owned *Feature*s are related to the owning *Classifier* by the *feature* relation. The *Operation*s in a *Classifier* are implemented by a set of other *Classifier*s, associated to the current by *realization*, the inverse relation is called *specification*. Relations are connected by the *participant* association.

One of the direct *Classifier* subtypes is **DataType**. A *DataType* is a type whose *Operation*s are pure functions and cannot change the state of the corresponding *DataValue* instance. The values of a *DataType* have no identity (are not objects). In this sense, *DataType*s are distinguished from *Class*es by their primitive values. The basic *DataType*s that are used in the UML have been presented in the Data Types package.

The internal structure of *Classifier*s are called the *Feature*s of the *Classifier*. *Feature*s describe both, the structure, and the dynamics of the *Classifier*.

> **Feature** is an abstract, direct *ModelElement* subclass. *Feature*s are used to characterize the structure and behavior of a *Classifier* or *Instance*.

A *Feature* may be an *Operation* or an *Attribute* of an *Interface*, a *Class* or a *DataType* that is encapsulated in these *Classifier*s. The *Classifier* that contains the *Feature* is specified by the *owner* association. *Feature*s are equipped with the *visibility* attribute, that defines if the *Feature* is accessible from outside the owning *Classifier*. *Feature*s, that concern the structural aspects are instances of *StructuralFeature*.

> **StructuralFeature** is a direct, abstract *Feature* subclass. *StructuralFeature*s are used to specify the structural properties of a *Classifier* or an *Instance*.
> In the standard metamodel, the only concrete subclass of *StructuralFeature* is **Attribute**.

An *Attribute* is owned by a *Classifier* or *Instance* and specifies the range of their values. The *type* association of *Attribute* specifies a *DataType* or a *Class*, whose instances provide the *values* of the *Attribute*. The *changeable* attribute specifies if the *value* of the *Attribute* may be modified after the creation of the *Object* owning the *Attribute*. The number of *DataValue*s for the *Attribute* contained in an *Instance* are specified by the *multiplicity* attribute.

The *Feature* metaclass possesses another direct subclass, the *BehavioralFeature* metaclass, whose subclasses are used to specify and realize the dynamical behavior of *Class*es.

> **BehavioralFeature** is an abstract, direct subclass of *Feature*. *BehavioralFeature*s are used to define dynamical aspects of a *Classifier* like *Operation*s that may be performed by the *Classifier* instances.

*BehavioralFeature*s may arise in form of *Operation*s and *Method*s. Both define the behavior that may be performed in a *Classifier*. While *Operation*s specify the behavior only conceptually, *Method*s must provide concrete procedural specifications, which must be executable at run-time.

> **Operation** is a direct *BehavioralFeature* subclass. *Operation*s are used to describe the behavior of the *Instance*s of a containing *Classifier*, that can be requested.

The specification of an *Operation* provides it with a *name* attribute, the used parameter and return types for its performance, and its *visibility* from outside the containing *Classifier*. The *ownerScope* attribute defines, if the *Operation* is applicable to instances, or the owning *Class* itself. The *isPolymorphic* association of an *Operation* describes whether the implementation of the *Operation* may be overridden by subclasses. The signature of an *Operation* is a set of *Parameter*s used in the specification of the *Operation*. The *parameter* association offers the signature as an ordered collection of relations to the demanded *Parameter*s.

> **Parameter** is a direct *ModelElement* subclass. A *Parameter* declares an argument that can be changed and passed or returned to or from an *Operation* or *Signal*.

A *Parameter* may include a unique name, a type or a direction of communication.

The second subclass of *BehavioralFeature* is the *Method* metaclass. *Method*s implement *Operation*s by specifying a procedure that realizes the result of the action of an *Operation*.

> **Method** is a direct *BehavioralFeature* subclass. A *Method* is connected to the *Operation* it implements by a *specification* association.

A *Method* and its associated *Operation* must be owned by the same *Classifier*. They have the same *visibility* and signature. The implementation of the *Method* is a *ProceduralExpression* that is specified in its *body* attribute.

Dependencies and associations

There are three different types of relationships in the standard UML-metamodel, namely, *Generalization*, *Dependency*, and *Association*. The *Dependency* and *Generalization* relations are direct subclasses of *ModelElement*, while the *Association* metaclass is a relationship that may be generalized. *Association*s may arise with additional properties in form of *AssociationClass*es.

> **Dependency** is a direct *ModelElement* subclass. A *Dependency* is used to manifest a directed relation between different *ModelElement*s. The functionality of the source of a *Dependency* relation (*client*) is influenced by the element corresponding to the target (*supplier*) of the relation.

In essence a *Dependency* relation implies that the implementation or realization of the depending element preassumes the realization of other elements of the model.

All semantical relations between *Classifier*s are given with the help of the *Association* metaclass. Its instances are tuples of *Instance*s of *Classifier*s.

> **Association** is a direct *GeneralizableElement* subclass. An *Association* relates a set of *Classifier*s. Related *Classifier*s correspond to *AssociationEnd*s owned by the *Association*.

*Association*s are contained in the *Namespace* of the connected *Classifier*. With respect to this *Classifier*, it has a unique *name* attribute in the *Namespace*. The connection of *Association*s and *Classifier*s are via *AssociationEnd*s. Every *Association* owns at least two *AssociationEnd*s.[4] The instances of *Association*s are called *Link*s.

> **AssociationEnd** is a direct *ModelElement* subclass. *AssociationEnd*s are owned by *Association*s. They carry information about which *Classifier* the *Association* connects, the *multiplicity* of the relation and the accessibility of the connection.

*AssociationEnd*s represent the connection points of an *Association* and a *Classifier*. The *targetScope* attribute states, if the connected element is an instance or the *Classifier* itself. The *AssociationEnd* may be traversed towards the *Classifier* when the *isNavigable* attribute is true. The *multiplicity* attribute defines how many instances are related at the specific *AssociationEnd*.

The **aggregation** and **composition** relations are realized as binary *Association*s, where exactly one *AssociationEnd* is specified by the *aggregation* attribute of the *AssociationEnd* metaclass. The values of the *aggregation* attribute are *aggregation*, *composition*, or *none*, denoting weak, strong, or not at all ownership, respectively. In a composition, the semantics of the whole are propagated to its parts, for example, destroying the whole results in the destruction of all parts. Consequently, a *ModelElement* may be the part of at most one composition, although the owning *ModelElement* may change. In a weak aggregation, the semantics of the whole are not propagated to its parts. Every aggregation is a transitive and antisymmetric relation of *ModelElement*s, leading thereby to a directed acyclic graph (DAG) of aggregations. Obviously, composition hierarchies are tree graphs.

Classes and interfaces

One of the most used metaclasses is the *Class* metaclass. It describes a set of objects that have the same *Feature*s, relations and semantics.

> **Class** is a direct *Classifier* subclass. A *Class* owns *StructuralFeature*s and *BehavioralFeature*s and defines the data structure of *Object*s.

A *Class* is the canonical *Namespace* of its *Attribute*s, *Association*s and *Operation*s. Elements that are contained in the class, like other *Class*es, *Interface*s, or role names, are defined with respect to this *Namespace*, too. All contained elements specify their *visibility*. In particular, the values of the contained *Attribute*s may be common to all instances of the *Class*, or individual for every instance. This is specified by the *ownerScope* attribute.

[4] Unfortunately, the figure on p.37 in [81] confuses the *AssociationEnd* with the *AssociationRole* metaclass.

The *Object*s of a *Class* must have values compatible to the *StructuralFeature*s and behave according to the *BehavioralFeature*s specified in the *Class*. *Class*es that enable the instantiation of *Object*s are called **concrete**, otherwise they are denoted **abstract**. Figure 5.10 shows the static semantics of the *Class* metaclass.

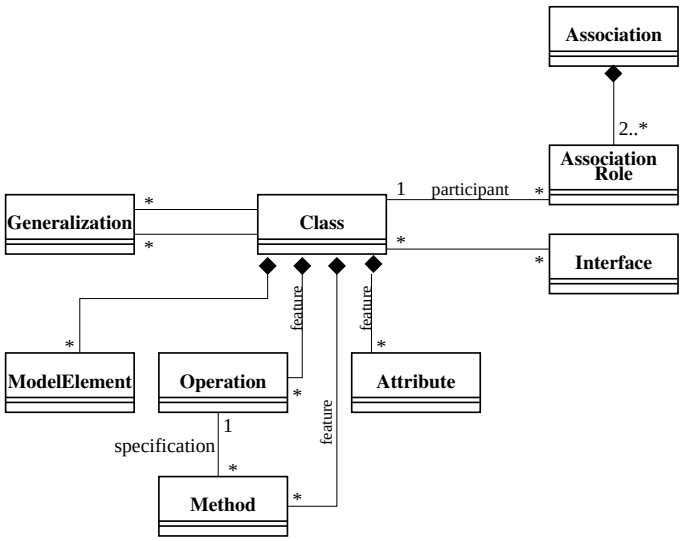


Figure 5.10: The *Class* metaclass and its associations.

A *Classifier* may realize a collection of *Interface*s. *Interface*s are partitions of the *Operation*s contained in a *Classifier* and describe a coherent behavior that is realized by the *Classifier*. This means, that all *Operation*s present in an *Interface* are contained in one and only one *Classifier*. However, the correspondence between *Interface*s and *Classifier*s is not one to one. A *Classifier* may realize more than one *Interface*, and an *Interface* may be offered by more than one *Class*.

> **Interface** is a direct *Classifier* subclass. *Interface*s own exclusively some *Operation*s of a *Classifier* describing thereby a coherent behavior of the realizing *Classifier* that is requestable from outside the *Classifier*.

*Interface*s may not have *Attribute*s, *Association*s or *Method*s and cannot contain *Classifier*s. They are *GeneralizableElement*s and transmit their *Operation*s to their heirs.

Associations equipped with features

To enable an *Association* to be equipped with further complex *Feature*s the metamodel offers a subclass of *Association*, the *AssociationClass*.

> **AssociationClass** is a subclass of *Class* and of *Association*. It owns a set of *Feature*s and of *AssociationEnd*s.

In a sense, *AssociationClass*es are *Association*s, which provide also *Class* properties. *AssociationClass*es are used to add relation-specific *Feature*s to an *Association*, which are not contained in the related *Classifier*s. Their semantics combine the *Class* and the *Association* semantics.

5.4.2 THE AUXILIARY ELEMENTS PACKAGE

The Auxiliary Elements package includes some further static concepts that extend the Core package. The Auxiliary Elements package contains the *ViewElement* metaclass, the concepts to describe the physical structure of a model, the definition of templates, and specializations of the *Dependency* metaclass.

Physical Structures

Many UML-models are implemented as software in computer networks. To be able to incorporate the physical as well as the software architecture in the UML-model, the *Node* and *Component* metaclasses are introduced. A **Component** is a direct subclass of *Class*. *Component*s are used to describe the physical package (software) of its *ModelElement*s. The *deployment* association connects it to its physical location, called *Node*. **Node** is a direct *Class* subclass. It represents physical objects (hardware), where *Component*s reside.

Templates

A *ModelElement* containing at least one unbound *Parameter* is called a *template*, or *parameterized element*. The *Parameter*s are given by the *templateParameter* association as an ordered list. A template does not provide a descriptor for a *Class* and is thereby an abstract *ModelElement*. The *Parameter*s of a template may be partially bounded. The well-formedness is not required unless a *Binding* relation occurs, that transforms all *Parameter*s into *Argument*s. The template is not a part of the final model, only bounded templates occur in the final model.

Dependencies

In addition to the properties introduced in the Core package, the Auxiliary Elements package extends this metaclass by the *owningDependency* and the *subDependency*. With these associations, a *Dependency* is a container of a set of semantically related dependencies. This is important, when *ModelElement*s are structured into *Package*s, or *Model*s. There are four *Dependency* subclasses included in the Auxiliary Elements package, *Refinement*, *Usage*, *Binding*, and *Trace*.

- **Refinement**
  Provides relations between semantically distinguished *ModelElement*s. A *Refinement* occurs as a *Dependency* from a *ModelElement* that is derived from its supplier. The specification of the *Refinement* content may be given by *Constraint*s, but is beyond

the scope of the UML. The *supplier*s and *client*s are connected to the *Refinement* by the *mapping* association.

- **Usage**
  A *Usage* represents the fact, that a *client* requires the *supplier*'s existence for its implementation or operability.

- **Binding**
  Relation between *templates* and *ModelElement*s. The *argument* association gives an ordered list of *Argument*s, that replace the corresponding *Parameter* in the template.

- **Trace**
  Used when different *Model*s of the same system are used to provide different views of a system. A *Trace* manifests a conceptual connection of *ModelElement*s in different *Models*, that specify the same part of the modeled system. The direction of a *Trace* is usually neglected.

The *Dependency* subclasses are applied inside the metamodel, too.

5.4.3 MODELING EXTENSIBILITY

Despite the large number of metaobjects in the UML-metamodel, it is not possible to provide complete semantics for every kind of modeling scenario that may arise. One way out of this dilemma is to allow the unrestricted extension of the metamodel by additional metaclasses. However, this would result very fast in the incompatibility of different UML-models, since many advanced concepts of the model would rely on model-specific, new metaclasses. Consequently, this would destroy one of the most important advantages of the UML, namely its role as a widely accepted standard for system specification and modeling.

To avoid such obstacles, the UML offers three standard extension mechanisms, which may be used to modify the metamodel softly. These mechanisms are given by metaclasses of the metamodel itself, namely by *Stereotype*s, *TaggedValue*s and *Constraint*s. These metaclasses apply to model elements, not to instances on the data & process level. Extension mechanisms specify how to modify the UML-metamodel in a way, that the resulting semantics fulfil the desired requirements. On the one side, one may introduce new types or equip existing types with required properties, on the other side one may modify more softly by changing the existing semantics of types by certain restrictions. The latter alternative is realized by *Constraint*s, which were introduced in the Core package. *Constraint*s restrict the instantiation of an instance of a *ModelElement*. Typical examples are the `'{disjoint}'` and the `'{overlapping}'` *Constraint*, that apply on *Generalization*. The former states, that an instance cannot originate in more than one subtype of a given type. The latter states that it can.

*Stereotype*s and *TaggedValue*s are defined in the Extension Mechanisms package. A *Tagged-*

Value is a simple extension mechanism, that permits the attachment of a semantical property to a given *ModelElement*.

> A **TaggedValue** is a pair `{tag=value}`, where `tag` is a name that is used as a pseudoattribute in a *ModelElement* to which the `value` is assigned uniquely.

There exists some standard predefined *TaggedValue*s in the UML-metamodel. Examples are the `'semantics'`, that applies to *Classifier*s and *Operation*s to specify their (dynamical) meaning. Another *TaggedValue* is `'location'`, which applies to *Classifier*s and *Component*s denoting that they are part of a specific *Component*, for example

`{location=server}`.

However, the interpretation of the tags and the specification of the corresponding values are outside the scope of the UML.

The *Stereotype* concept is in essence an elegant way to mark *ModelElement*s while leaving their structure (attributes, operations, relations) invariant. *Stereotype*s are used to softly modify the meaning of the elements they are attached to, by requiring for example additional *Constraint*s or *TaggedValue*s. *Stereotype*s provide a frequently used extension mechanism for a flexible modification of the UML-metamodel.

> **Stereotype** is a direct *GeneralizableElement* subclass. *Constraint*s and *TaggedValue*s may be associated with a *Stereotype*. The *Stereotype* is associated with a *ModelElement* which is called its *baseClass*.

The name of the extended metaclass is stored in the *baseClass* attribute of the *Stereotype*. The corresponding association role is called *extendedElement* and connects the corresponding *Stereotype* and *baseClass* instances. Some examples are:

- `<<system>>`[*Package*]
  Specifies a *Package* that represents the entire modeled system as a collection of *Model*s. The different *Model*s need not necessarily be disjoint and describe different views of the modeled system.

- `<<type>>`[*Class*]
  Is used for abstract *Class*es to specify a domain of *Object*s and their *Operation*s without specifying their implementation. A `<<type>>` may not contain any *Method*s.

- `<<implementationClass>>`[*Class*]
  Opposite to `<<type>>`, this *Stereotype* specifies the implementation of a *Class* in some programming language.

- `<<powertype>>`[*Classifier*]
  A *Classifier* that is a metatype, whose instances are subtypes of another type.

- `<<metaclass>>`[*Classifier*]
  Specifies a *Classifier* whose instances are exclusively *Class*es.

- `<<inherits>>`[*Generalization*]
  Describes *Generalization*s which violate subtyping. Instances of a subtyping *Classifier* are not substitutable for the supertype instances.

- `<<stereotype>>`[*Classifier*]
  Specifies *Classifier*s that may be applied as *Stereotype*s to *Classifier*s.

Since *Stereotype* may be generalized, a subtype inherits all *Constraint*s and *TaggedValue*s of its supertype. Subtyped *Stereotype*s must be applied to the base class of their ancestors.

5.5 MODELING DYNAMICS

The dynamics of a UML-model is described by the concepts of the *Behavioral Elements* package. This package contains the *Common Behavior* package, the *State Machines* package, the *Collaborations* package and the *Use Cases* package.

5.5.1 THE COMMON BEHAVIOR PACKAGE

The *Common Behavior* package underlies all other packages of the Behavior Elements package. It defines many instance-analogons of the type/instance dichotomy, and offers the concepts for the elementary communication of objects. Moreover, the foundations for more advanced dynamical considerations, like state machines, collaborations and use cases are provided. The logical hierarchy of the Behavior Elements package is given in fig 5.11

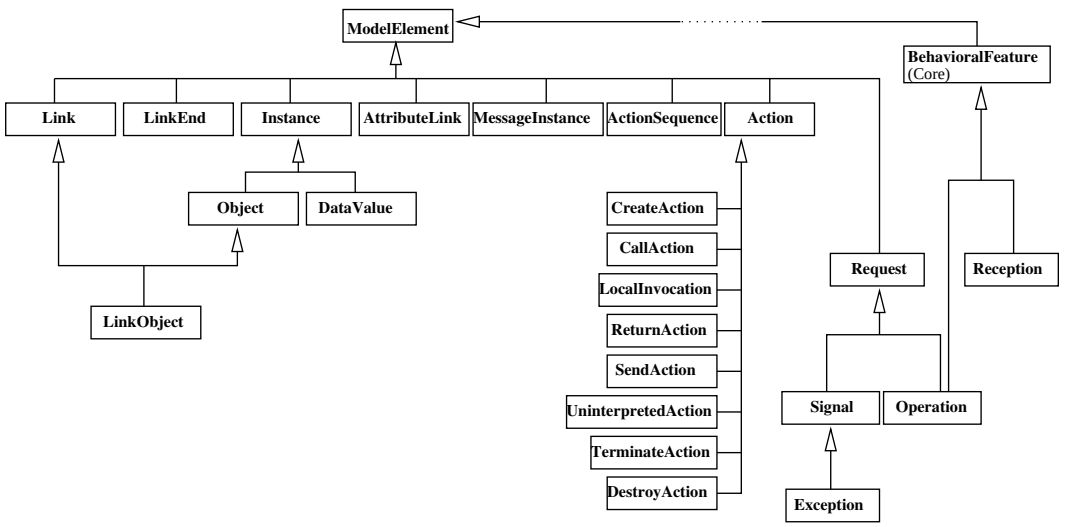


Figure 5.11: The generalization hierarchy of the *Common Behavior* package. The defined metaclasses underly the metaclasses in the *Behavior Elements* package.

One of the most important metaclasses in the Common Behavior package is the *Instance* metaclass. It builds the type/instance analogon to the *Classifier* metaclass in the Foundation package. An *Instance* must originate in a *Classifier* and enables the application of *Operation*s specified therein. The effect of an applied *Operation* is stored in the *Instance*'s state. The state of an *Instance* describes a specific set of attribute values and links.

> **Instance** is a direct, abstract subclass of *ModelElement*. The *classifier* association connects an *Instance* with the set of *Classifier*s, that specify its structure and behavior.

The attribute values of an *Instance* are given by a set of *AttributeLink*s, connected to the *Instance* by the *attributeLink* association. The *Association*s of the corresponding *Classifier*s result in *Link*s connected to the *Instance* by matching *LinkEnd*s. The *LinkEnd*s are given by the *linkEnd* association. The collection of *AttributeLink*s and *Link*s defines the specific state of the *Instance*.

DataValues, Objects and Links

Following the type/instance dichotomy, the Common Behavior package contains many metaclasses, that build the instance metaclass of this pairing. This is the case for the pairs *Class/Object*, *Association/Link* and *DataType/DataValue*.

The type/instance analogon of the *DataType* metaclass is the *DataValue* metaclass.

> **DataValue** is a direct *Instance* subclass. A *DataValue* contains exactly one state, *Operation*s applied to a *DataValue* simply map to another *DataValue*.

A *DataValue* instance has no identity and cannot change its type. Since *Operation*s act on *DataValue*s as simple functions or queries without side effects, the state of a *DataValue* cannot change, too. Usually, *DataValue*s are used in the information structure of *Classifier*s as values for the contained *Attribute*s. The corresponding connection is provided by the *AttributeLink* metaclass.

> **AttributeLink** is a direct *ModelElement* subclass. An *AttributeLink* contributes to the state of an *Instance* by holding the value of an *Attribute*.

The two associations *value* and *attribute* connect the *AttributeLink* to the *DataValue* and to the *Attribute*, respectively. *DataValue*s contain no *AttributeLink*s.

The *Association* metaclass and the *Link* metaclass build another type/instance pair. *Link*s connect *Instance*s, matching thereby *Association*s that connect the corresponding *Classifier*s.

> **Link** is a direct *ModelElement* subclass. A *Link* owns a set of *LinkEnd*s that match the corresponding *AssociationEnd*s.

*Link*s manifest the communication channels of *Instance*s. A *Link* is determined uniquely, once an *Association* and the connected *Instance*s are specified. *Link*s are connected to *Instance*s by *LinkEnd*s.

> **LinkEnd** is a direct *ModelElement* subclass. A *LinkEnd* provides the connection of an *Instance* and a *Link*.

LinkEnd is the type/instance analogon to the *AssociationEnd*s metaclass. Another specialized *Instance* is the *Object* metaclass, which is the type/instance analogon to *Class*. An *Object* is an identifiable *Instance*, whose structure is determined by the structure of the *Class* from which it originates. Different *Object*s originating in the same *Class* possess their own *AttributeLink*s, which underly the *multiplicity* attribute of the corresponding *Attribute*.

> **Object** is a direct subclass of *Instance*. An *Object* originates in a *Class*, which describes its structure and its associations.

It is important to note, that the behavior of an *Object* is specified in the corresponding *Class*. Since the set of *Class*es, from which an *Object* originates may change dynamically, the behavior of an *Object* is not invariant through its lifetime. New *Class*es add new *Feature*s to the structure of the *Object*, while detaching *Class*es removes *Feature*s from the *Object*'s structure.

As for *AssociationClass*es, the fusion of the properties of *Link*s and *Object*s is realized in the *LinkObject* metaclass. Roughly spoken, a *LinkObject* is a *Link* that is also an *Object*.

> **LinkObject** is a direct subclass of *Link* and *Object*. A *LinkObject* is a *Link*, that contains a set of *AttributeLink*s and *Operation*s.

A *LinkObject* must originate at least from one *AssociationEnd*, building thereby its type/instance analogon. Of course, its *AttributeLink*s and *Operation*s must match the *Attribute*s and *Operation*s of the *AssociationEnd*.

Communication of Instances

Here the metaclasses *Request*, *Signal*, *Exception* and *MessageInstance* are introduced.

To describe the concrete communication between *Instance*s, one uses the *Request* metaclass.

> **Request** is a direct, abstract subclass of *ModelElement*. A *Request* specifies the stimulus that is sent to an *Instance*.[5]

[5]The definition of the *Request* metaclass in [81] defines it as a direct subclass of *BehavioralFeature*. This conflicts the definition of the *Signal* metaclass, since a *Signal* represents a communicated stimulus, rather than a *BehavioralFeature* of a *Classifier*. Therefore *Request* is defined here according to the class diagram on p. 66 of the same document.

A *Request* has two direct, concrete subclasses, the *Signal* and the *Operation* metaclasses. *Request*s that occur as *Operation*s specify the execution of an *Operation* in the receiving *Instance* and can be asynchronous or synchronous. In the former case, the sending *Instance* does not pause to wait for responding messages, whereas in the latter case it does. In contrast, *Signal*s are defined as specific, asynchronous *Request*s for the communication of *Instance*s. They trigger reactions in the receiver, without replying to the sender.

> **Signal** is a direct subclass of both, *Request* and *GeneralizableElement*. A *Signal* aggregates a set of *Parameter*s.

*Signal*s are interpreted in the receiving *Instance* by *StateMachine*s. To be able to treat execution faults in a run-time UML-model, a specific *Signal* called *Exception* is introduced. It is a direct *Signal* subclass, associated with a *BehavioralFeature*, which raises it in case of faults. In general, *Classifier*s, that are able to handle a specific *Signal*, are connected to the *Signal* by the *reception* association. The connected *Classifier*s contain a *Reception*, which summarizes the behavior of the *StateMachine* in the *Classifier*, that executes the *Signal*.

> **Reception** is a direct *BehavioralFeature* subclass. A *Reception* declares that a *Classifier* is able to react to a *Signal*.

The *isPolymorphic* attribute states, if the specific response to a *Signal* may be overridden in subclasses of a *Classifier* containing the *Reception*.

A concrete communication of two *Instance*s is described by a *MessageInstance*. It is the type/instance analogon to *Request*.

> **MessageInstance** is a direct *ModelElement* subclass. A *MessageInstance* arises as an instance of a *Request*.

The *sender* association of a *MessageInstance* connects it to the *Instance*, that has sent the *MessageInstance*, the *receiver* association to the receiving *Instance*. The *argument* association connects a *MessageInstance* to a sequence of *Instance*s, that build the *Argument*s of the *MessageInstance*. A *MessageInstance* that originates in a *CallAction* causes the execution of the *Method*s described in the full descriptor of the *Operation*, which was specified by the *CallAction*.

Actions

*Action*s describe abstractions of mechanisms, that change the state of a model by sending messages to objects, or by modifying the value of an attribute.

> **Action** is a direct, abstract *ModelElement* subclass. The subclasses of *Action* define executable statements in the model.

*Action*s are always executed in *Instance*s. An *Action* may specify a *target* attribute showing recipients of a specific *Request*, and an *actualArgument* association declaring a sequence

of *Expression*s that show the kind of *Argument*s needed in the execution of the *Action*. *Argument*s are aggregated by *Action*s. An *Argument* is connected to its *Expression*s by the *value* association. Every *Expression* determines an *Instance* when evaluated. *Action*s arise both, in asynchronous and synchronous versions. Usually, *Action*s are aggregated by *ActionSequence*s.

> **ActionSequence** is a direct *ModelElement* subclass. The sequence of *Action*s is given by the *action* associations of *ActionSequence*.

An *ActionSequence* describes the behavior of a *State*, or a *Transition* that owns it. The behavior is specified by the sequence of *Action*s contained in the *ActionSequence*.

Since *Action* is an abstract metaclass, there exist several subclasses, that specialize it. The subclasses define the specific result of their execution. In a sense, the *Action* subclasses describe the standard behavior, that may be performed without further specification of model details.

> **SendAction** creates the sending of a *Signal* to some set of receivers. It is an asynchronous *Action*.
> **CallAction** creates a *MessageInstance* that causes the invocation of an *Operation* by mediation of a *StateMachine* in the receiver. *CallAction*s may be synchronous or asynchronous.
> **LocalInvocation** causes the execution of an *Operation* in the *Instance* itself. It is distinct from a *CallAction* sent to the *Instance* itself, since it invokes a local *Operation* without entering a *StateMachine*.
> **TerminateAction** causes an *Instance* to destroy itself.
> **CreateAction** causes the creation of an *Instance* according to the structure defined in by set of *Classifier*s.
> **DestroyAction** results in the destruction of the *Instance* receiving the message.
> **ReturnAction** causes the return of a value to a caller *Instance*.
> **UninterpretedAction** any *Action* that has no UML-specific interpretation.

The *Action* metaclass contains a *recurrence* metaattribute specifying how many *MessageInstance*s should be dispatched. In addition the *recurrence* metaattribute defines a collection of *MessageInstance*s that should be dispatched sequentially, or parallel.

5.5.2 SOCIETIES OF OBJECTS: COLLABORATIONS

Once the structure of a UML-model is given by *ModelElement*s of the Foundation package, and its elementary dynamics by the *ModelElement*s of the Common Behavior package, it is necessary to specify how these constituents collaborate in order to produce more complex behavior. In the UML, a collection of *Classifier*s, *Association*s and *Signal*s, that work together by interactions to provide some specific behavior, is called a *Collaboration*. A *Collaboration* defines the context of a specific interaction of *ModelElement*s. For this, *Classifier*s and *Association*s in a *Collaboration* are equipped with roles specifying their tasks in

an interaction. The structure of a *Collaboration* is then the logical view of all its containing *ClassifierRole*s and *AssociationRole*s. Its behavior is specified by the *Interaction*s it contains.
The generalization hierarchy of the metaclasses are shown in fig. 5.12. To avoid graphical overloading some associations are not presented.

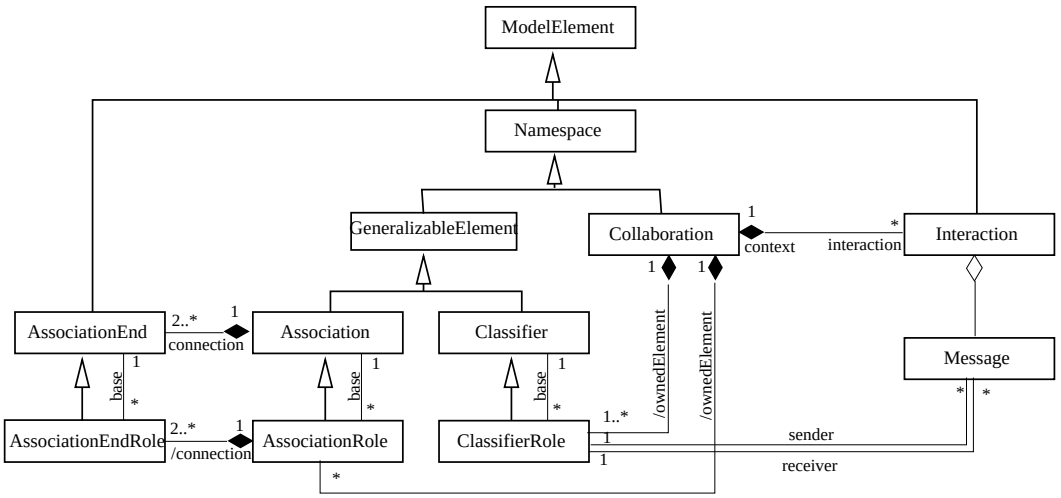


Figure 5.12: The metaclass-hierarchy of the Collaboration package. It defines *Collaboration*s, *AssociationRole*s, *AssociationEndRole*s, *ClassifierRole*s, *Interaction*s and *Message*s.

*Collaboration*s define the context in which a part of the behavior of a model is realized. They do not directly contain *Classifier*s or *Association*s. Rather, they contain parts of these elements, that participate in the specific *Collaboration*s. The part of a participating element is its role in the *Collaboration*. Roles specify the concrete subset of *Feature*s and *Association*s that a participating element uses to enable the required interactions of the *Collaboration*. There are three kinds of roles relevant in a *Collaboration* context, *AssociationEndRole*s, *AssociationRole*s and *ClassifierRole*s.

> **AssociationEndRole** is a direct *AssociationEnd* subclass. An *AssociationEndRole* is part of an *AssociationEnd* and specifies the connection of a matching *AssociationRole* to a *ClassifierRole*.

An *AssociationEndRole* specifies the usage of an *AssociationEnd* in a *Collaboration*. It must match to an *AssociationRole*.

> **AssociationRole** is a direct *Association* subclass. It is a composition of at least two *AssociationEndRole*s. An *AssociationRole* is a part of an *Association* contained in a *Collaboration*.

*AssociationEndRole*s and *AssociationRole*s are used to connect *ClassifierRole*s that participate in a *Collaboration*.

> **ClassifierRole** is a direct *Classifier* subclass. A *ClassifierRole* is a part of a *Classifier* that is contained in a *Collaboration*. The *Feature*s of a *ClassifierRole* build a subset of the *Feature*s of the base *Classifier*. A *ClassifierRole* is connected to *AssociationEndRole*s contained by *AssociationRole* which match the *Association*s of the base *Classifier*.

Every role in a *Collaboration* specifies a *multiplicity* attribute, which defines how many instances of the role participate in the *Collaboration*. Further, every role possess a *base* association specifying the *Classifier*, *Association*, or *AssociationEnd* the corresponding role refers to. Moreover, the *ClassifierRole* possesses an *availableFeature* association, which specifies the *Feature*s of the *ClassifierRole*. These *Feature*s must build a subset of the *Feature*s of the *base Classifier*, no own *Feature*s are allowed. Obviously, by its definition more than one role of a *base* element may occur in a *Collaboration*.

The dynamic aspects of *Operation*s, *Classifier*s, or *UseCase*s are present in a model in form of *Interaction*s. *Interaction*s model the communications within a *Collaboration*. The corresponding roles in the *Collaboration* communicate with each other by *Message*s.

> **Message** is a direct *Signal* subclass. A *Message* specifies the usage of a *Signal* in an *Interaction* of *Classifier*s of a *Collaboration*. A *Message* specifies its order in the sequence of *Message*s of the *Interaction*, the roles of its sender and receiver, and the dispatching *Action*.

To specify the usage of a request, a *Message* is associated to several elements. The *sender* and *receiver* associations specify the *Instance* role, that sends, or receives the *Message* respectively. Both must be contained in the *Collaboration*. The *activator* association connects it to the *Message* which called the *Operation* containing the current *Message*. The *predecessor* association shows the *Message*s, that must be completed for the current *Message* to be enabled to execution. Both, *activator* and *predecessor* must be contained in the *Interaction* of the current *Message*. Usually, *Message*s form a sequence in an *Interaction* to specify the execution order.

> **Interaction** is a direct *ModelElement* subclass. An *Interaction* is owned by a *Collaboration* and contains the set of *Message*s, that describe the communication of *Instance*s of the *ClassifierRole*s, participating in the *Collaboration*.

The *context* association of an *Interaction* connects it with the *Collaboration* that defines its context, whereas the *message* association specifies its containing *Message*s.

The metaclasses introduced above play together in the definition of a *Collaboration*. *Collaboration*s unify the structural specification of participating roles, and the corresponding dynamical *Interaction*s that specify their communications.

> **Collaboration** is a direct subclass of *Namespace*. A *Collaboration* contains *Interaction*s, *ClassifierRole*s and *AssociationRole*s representing the parts of the

> *base Classifier*, *base Associations*, and base *Signals* and the way they participate in the *Collaboration*. *Collaborations* specify the realization of *Classifiers* or of *Operations*.

There are two main usages of *Collaborations* in the UML. First, *Collaborations* are used to define the realization of a *Classifier* or an *Operation*. A *Collaboration* specifies how these *ModelElements* realize their behavior based on the interaction of other *ModelElements*. For example, an *Operation* is realized as a *Collaboration* of *Methods*, whereas a general *Classifier* may reflect a *Collaboration* of any *Classifier* in the system. Second, a *Collaboration* may be used to refine a larger scale *Collaboration*.

> *Therefore, by composition,* Collaborations *provide a natural multiscaling in UML-models.*

The composition property is explained in the next subsection.

5.5.3 COMPOSITION OF COLLABORATIONS

One of the most important properties of *Collaborations* is their **composibility**. Two or more *Collaborations* may be composed into a superordinate *Collaboration*, which unifies their functionalities. The composition of *Collaborations* is frequently used in refining *Classifiers* by *Collaborations* of subordinate *Classifiers*. To compose two or more *Collaborations*, one distinguishes the roles contained in distinct *Collaborations* and corresponding to identical *base* elements. By union of the features and associations described by the roles one gets a new role that unites all properties and provides therefore the 'fusion-points' for the superordinate *Collaboration*. This is shown schematically in fig. 5.13. *The* Collaborations *are represented by ToCS-systems consisting of* ClassifierRoles *(ToCS-objects) and* Messages *(ToCS-links).* The *Message* ordering is disregarded.

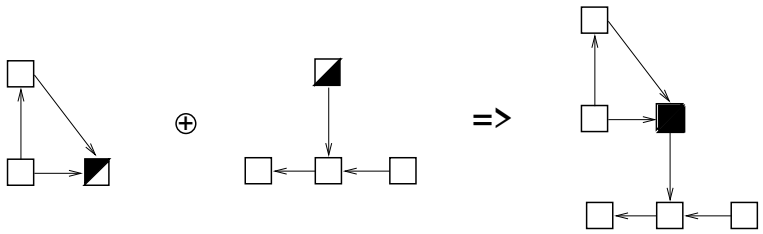


Figure 5.13: Composition of two *Collaborations*. The half-filled boxes represents *ClassifierRoles* that correspond to the same *base*. Their features and associations are combined into the new *ClassifierRole* that fulfils the requirements of both *Collaborations*.

Of course, for the complete definition of the composed *Collaboration*, a new sequence of the contained *Messages* must be specified in the new *Interaction*.

For example, in refining a *Classifier* by a *Collaboration* of other *Classifiers*, there exists at

least one *Classifier*, which has *ClassifierRole*s in both subordinate *Collaboration*s. This relies on the fact, that there must be at least one *Association* connecting both *Collaboration*s, that matches existing *AssociationEndRole*s. This *Association* provides the corresponding *AssociationRole*s, on which the *Message* relies. Since the *sender* of a *Message* must be contained in the *Collaboration* too, this provides a standard fusion point.

5.5.4 STATECHARTS AND OBJECT STATE MACHINES

Once the structural aspects of a system are modeled and its elementary interactions are analyzed by *Collaboration*s, the question arises how the internal behavior of *ModelElement*s should be expressed and connected to processes involving many elements of the model. In general, the specification of the behavior of an entity describes the required inputs, the produced outputs, and the various detailed relations between them [99]. One major group of behavioral specification are state-related techniques, that is, approaches that focus on states and state changes.[6] The most important state-related approaches are based on finite state machines.

> A **finite state machine** (FSM) is a hypothetical machine that can exist in only one of a finite number of states at a given moment [70].
> A FSM proceeds in well-defined, discrete steps from one of its finite number of configurations or states to another [100].

There are several ways to specify a FSM. One way to approach a FSM is through state-event matrices, which define a set of states and their reaction on a finite number of events by a finite matrix. Another way is to specify the FSM as a graph, consisting of states and corresponding state transitions. In this approach, one has to decide where to attach the operations of the system. A graph representing a FSM by associating operations to the edges of the graph is called a **Mealy machine**, whereas if the operations are attached to the nodes the graph is called a **Moore machine**. Obviously, both approaches are mathematically equivalent. In the UML, Mealy and Moore machines are used in a hybrid form known as **statecharts**. Statecharts are graphs that combine Mealy and Moore machines. They were originally developed by Harel in 1987 [77] and have been extended to incorporate object-oriented concepts by Harel and Gery in 1997 [62]. In the latter form, statecharts can be implemented within classes, the relevant operations are then given by the operations in the classes. Therefore, statecharts can readily be mapped to object-oriented programming languages (and vice-versa). Another advantage is, that statecharts can easily be combined to larger statecharts. A schematic statechart is shown in fig. 5.14.
In general, FSMs are particularly useful when the complex behavior of a single object is modeled and when a small number of objects interact in event-driven, single threaded manner. The main disadvantages rely on the somewhat complicated modeling of multiple threaded systems and of interactions of a large number of objects. In addition, systems allowing many states at one moment may lead to unreadable FSM notation.

[6] Other techniques like decision-based and language based specifications are not relevant in the UML.

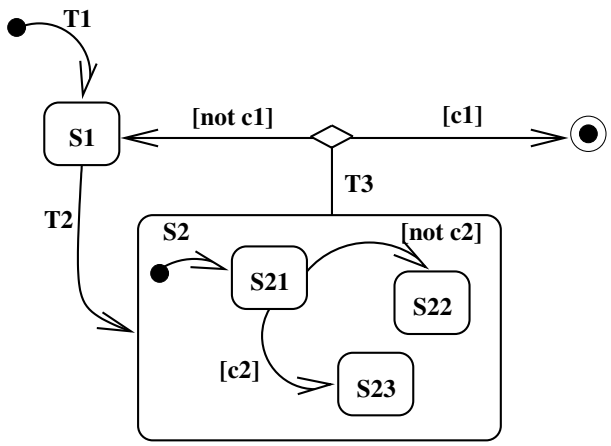


Figure 5.14: An abstract statechart. The dot denotes the initial state, the dot in the circle the final state. S1 is a simple state, whereas S2 is a composite state, containing the substates S21, S22 and S23. The initial state is changed to S1 by the transition T1, followed by T2 after completion of internal operations. The guards [c1], [not c1], [c2] and [not c2] are conditions whose validity designate the enabled transitions. The diamond splits the transition T3 according to its guards.

Object State Machines

In the UML, statecharts are used in a slightly modified form as object state machines. The main differences are the absence of actions, conditions and events as predefined in statecharts, the restricted transition composition in object state machines, the supporting of synchronous communication between state machines, and the zero-time assumption of statecharts. The zero-time assumption means that the transitions are assumed to take zero execution time. In object state machines this assumption is relaxed. The metaclasses that build the logical foundation of object state machines are contained in the *State Machines* package. The generalization hierarchy of the package content is shown in 5.15.
An object state machine describes the legal ordering of all states that an object may pass during its 'life'. Expressed as a graph, a state machine consists of a number of nodes representing states and edges representing transitions. The nodes being the sources or targets of transitions must match the state that are changed by the transition. In the UML, these nodes are described by the *StateVertex* metaclass.

> **StateVertex** is a direct *ModelElement* subclass. The *outgoing* and *incoming* associations specify *Transition*s that depart, or enter the *StateVertex*, respectively.

The *StateVertex* metaclass is used in the UML as the supertype for all possible nodes in a graph representing a FSM. Before passing to the definition of the *StateVertex* specializations and *Transition*s, the *Event* metaclass is introduced. An *Event* is a special occurrence, that is localized in space and time within a system. Usually *Event*s cause the activation of some *BehavioralFeature*s in an *Object*.

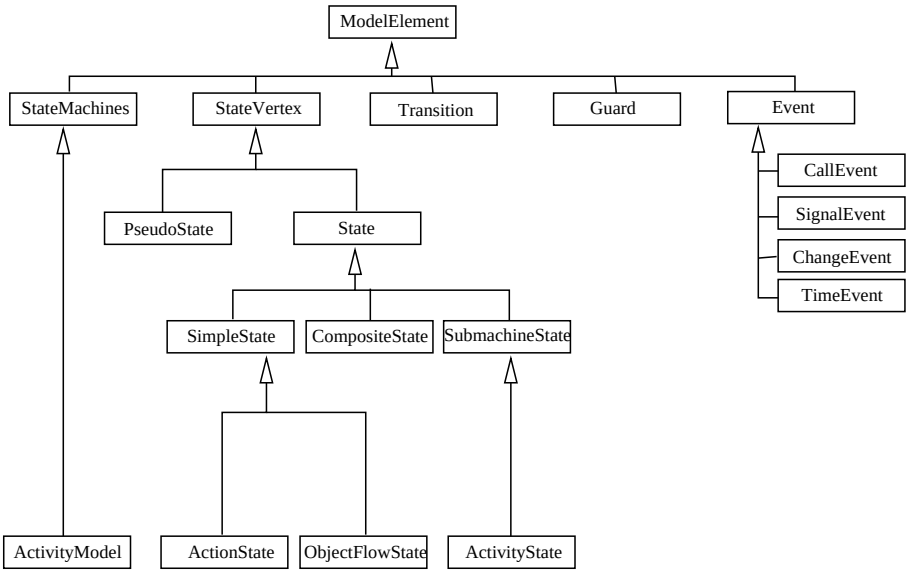


Figure 5.15: The metaclasses of the State Machines package. The four bottom metaclasses are used in activity models, which are special extensions of *StateMachine*s without individual dynamic semantics.

> **Event** is an abstract, direct *ModelElement* subclass. *Event*s are weakly aggregated within *Transition*s by the *trigger* association.

There are four concrete direct subclasses of *Event* in the metamodel.

> A **SignalEvent** denotes those *Event*s are caused by a *Signal* that is received by an *Object*. The *Signal*, whose reception may trigger a *Transition* is given by the *signal* association.
> A **CallEvent** is an *Event* that represents the reception of an *Request* which invokes an *Operation*. The *operation* association specifies which *Operation* should be invoked.
> A **ChangeEvent** is an *Event* that occurs when some values of some attributes or relations change. The *changeExpression* attribute is a boolean expression that specifies the conditions for the *ChangeEvent* to occur. *ChangeEvent*s are passive, in the sense that the run-time system detects the validity of the *changeExpression* and causes the *ChangeEvent*. In contrast to *Guard*s, a *ChangeEvent* triggers a *Transition*.
> A **TimeEvent** is an occurrence, that is caused by the expiration of a specific time deadline. The deadline is specified in the *duration* attribute.

In a state machine, *Event*s are used to trigger *Transition*s of one *StateVertex* to another. *Transition*s are binary relations of *StateVertex* instances. They are owned either, by *State*s as internal transitions, or by *StateMachine*s as compound transitions. A *Transition* represents

the transformation of a *State* into another one as a reaction on an *Event*. The transformation is realized by specific *Action*s.

> **Transition** is a direct *ModelElement* subclass. The *source* and *target* associations define the *StateVertex* that fires, and the one that receives the results of a *Transition*. *Transition*s are activated by specific *Event*s which are given by the *trigger* association. The *Transition* is enabled if and only if the condition associated to the *Transition* by the *guard* relation is fulfilled.

The *source* and the *target* of a *Transition* may be identical. The *effect* association describes an *ActionSequence* that constitutes the performance of the fired *Transition*.

*Transition*s may be enabled or disabled, according to the validity of certain conditions called *Guard*s.

> **Guard** is a direct *ModelElement* subclass. The boolean expression that specifies the condition that the *Guard* states on the *Transition* is stored in the *expression* attribute.

To avoid inconsistencies, *Guard*s should not be used for further side effects.

The *StateVertex* instances that *Transition*s connect, occur as *State*s or *PseudoState*s. A *State* is a situation in the 'lifetime' of an *Object*, that is defined by the validity of certain conditions, the performance of an activity, or by the *Object* waiting for an *Event* to occur.

> **State** is a direct *StateVertex* subclass. A *State* represents nodes in the state machine graph, which correspond to a possible situation in an *Object*'s lifetime.

The *entry* attribute of a *State* designates an *ActionSequence*, that is executed when a *Transition* enters the *State*. The *exit* attribute specifies an *ActionSequence*, executed when the *State* is leaved by an *Transition*. Both *ActionSequence*s are considered as atomic actions and cannot be avoided. *State*s may contain *Transition*s that occur only inside the *State*, that is, they do not invoke the *exit* or *enter ActionSequence*s and therefore cannot change the *State*. These *Transition*s are given by the *internalTransition* association. A *State* that is entered by a *Transition* is called an active *State*. The *State* becomes inactive when it is exited by the *Transition*.

The *State* metaclass is specialized to provide the semantics for atomic and composite states. There are three direct subclasses of *State*, namely *SimpleState*, *CompositeState*, and *SubmachineState*.

> **SimpleState** is a direct *State* subclass. *SimpleState*s cannot contain any further features, in particular they possess no substates.

To compose *SimpleState*s into higher-level *State*s, the *CompositeState* metaclass is used.

> **CompositeState** is a direct *State* subclass. A *CompositeState* strongly owns a collection of *State*s specified by the *substates* association, the *CompositeState* is connected to this association by the *parent* role.

By definition of strong ownership, a *State* contained in a *CompositeState* cannot be directly included in any other *CompositeState*.

A *CompositeState* may allow the decomposition of its internal structure into orthogonal and conjunctive components. A collection of *State*s is called **orthogonal**, if two included *State*s can be simultaneously active, in other words, if they are 'independent'. The *CompositeState* allowing an orthogonal decomposition is specified by the validity of its *isConcurrent* attribute, which is boolean. If true, the *CompositeState* contains at least two *CompositeState*s as its *substates* and is called a concurrent *State*. Conversely, a *CompositeState* that is itself contained in a concurrent *CompositeState* as an orthogonal component is specified by its *isRegion* attribute. As for general *State*s, a *Transition* that enters a *CompositeState* activates it. If the active *CompositeState* is a concurrent *State*, all of its *substates* are simultaneously active, since the entering *Transition* enters all of its regions, too. If it is non-concurrent, one and only one of its *substates* is active. There are three different ways for a *Transition* to enter a non-concurrent *CompositeState*, either by entering its initial *StateVertex*, or by entering an explicitely specified *substate*, or by entering a history state. In case of *deepHistory* the *Transition* enters lower-level history states recursively. In a similar way, a *Transition* may exit a concurrent, or non-concurrent *CompositeState*.

The last direct *State* subclass is *SubmachineState*. In essence, a *SubmachineState* is a *CompositeState*, being distinguished only by its semantics as a nested *StateMachine* included in another *StateMachine*.

> **SubmachineState** is a direct *State* subclass. A *SubmachineState* is associated to a specifying *StateMachine* by its *submachine* association.

*SubmachineState*s are introduced to support the modularity principle inside *StateMachine*s, and to improve reusability. Modularity is achieved since a complex *StateMachine* can be decomposed into more elementary *SubmachineState*s. All *CompositeState* semantics holds for *SubmachineState*s, too.

Not every *StateVertex* in a state machine represents a 'proper' *State*. Some *StateVertex* instances are used mainly to construct more complex *Transition*s, called compound *Transition*s, by branching, or unifying simple *Transition*s. To enable these tasks, some predefined transient states called *PseudoState*s are introduced, representing for example branches, or joins.

> **PseudoState** is a direct *StateVertex* subclass. *PseudoState*s are *StateVertex* instances that do not correspond to proper *State*s of *Object*s. The *kind* attribute classifies the *PseudoState* due to its usage and semantics.

There are seven possible values for the *kind* attribute, which may be classified in three groups.

- *initial / final*
  The *initial PseudoState* is used as the *source StateVertex* of the *Transition* that destinates in the 'default' *State*. For every *StateMachine* or *CompositeState* there is at most one *initial* vertex. *Initial* vertices have no *incoming Transition* and at most one *outgoing Transition*.
  The *final PseudoState* represents the termination of its owning *CompositeState*. In this sense, it is not considered as a transient state. If the parent *CompositeState* is the *top State* in a *StateMachine*, the *Object* described by the *StateMachine* reaches its 'deadline', otherwise the terminate *Transition*s are enabled.

- *join / fork / branch*
  A *join* represents a transient *StateVertex* that unifies *Transition* segments directly coming from a collection of orthogonal *State*s. The *incoming Transition*s cannot be conditioned by any *Guard*s. The *outgoing Transition*s are not fired by explicit *trigger*s.
  Conversely, *fork*s and *branch*es split a simple *incoming Transition*. *Fork*s represent transient vertices that connect exactly one *incoming Transition* from a *State* with a collection of*Transition*s to at least two orthogonal *State*s. A *outgoing Transition*s must not have any *Guard*s.
  A *branch* is a transient *StateVertex* that splits a unique *incoming Transition* into at two or more *outgoing Transition*s that target a set of orthogonal *State*s. The splitting designates the enabled *outgoing Transition*s by *Guard*s. Being a transient vertex, a *branch* needs no *trigger*s for its *outgoing Transition*s.

- *history*
  During the lifetime of an *CompositeState* it may be entered and exited by *Transition* more than one time. When not further specified, each time a *CompositeState* is entered through its *initial* vertex. However, sometimes it is necessary to store the latest active *substate* of the *CompositeState*'s last active period. This *State* is referenced by so-called *history* vertices. There are two kinds, the *deepHistory* and the *shallowHistory* vertex. The former remembers the most recent *substate* and its own *history* vertices, whereas the latter does not not store lower *substates*. A *CompositeState* can possess at most one *history* vertex, while every *history* vertex have at most one *outgoing Transition*.

The *join*, *fork*, and *branch* vertices are primarily used to compose simple *Transition* into compound *Transition*s. *Simple Transition*s connect *State*s. If more that three *Transition*s are connected at a *PseudoState*, the collection of *Transition*s combined at the *join*, *fork*, or *branch* vertex are called a *compound Transition*. As for simple ones, compound *Transition*s are atomic executions and possess at most one *trigger*. A simple or compound *Transition* T is enabled to fire, if the following conditions are fulfilled

- All *source*s of T are active.

- An arriving *Event* matches the *trigger* of T.
- A path of simple *Transition* segments from the *source*s of T to the *target*s of T, whose *Guard*s are fulfilled, exists. (Only relevant for compound *Transition*s.)

To support modularity, compound *Transition*s should be analyzed within the lowest *CompositeState* that contains them. In addition, to specify the execution of compound *Transition*s, the notion of *source* and *target* must be slightly generalized. Respecting *CompositeState*s, the *State* that is exited by a compound *Transition* is called its *main target*, while the *State* that it enters is called its *main target*. By definition, *main source* and *main target* may be composite. Furthermore:

> *For a* Transition T*, its* **Least Common Ancestor** LCA*[*T*] is defined as the lowest* CompositeState *that contains all explicit and enabled* source*s and* target*s of* T*. The* LCA *is used to encapsulate the analysis of a set of compound* Transition*s.*

Examples

In the statechart of fig. 5.14, suppose, that the *Guard* [c2] is valid. For the *Transition* from the nested *State* S21 to S23, the *main source* is S21, the *main target* is S23. The *LCA* of the *Transition* is the *CompositeState* S2.
Assuming the validity of the *Guard* [not c1] for the compound *Transition* T3 (branch), the *main source* is the composite S2, the *main target* is the *State* S1. However, note that *LCA*[T3] is given by the parent composite state of the whole statechart.

For the *main source* and *main target* in a *LCA*, the following remark holds:

> **Remark:**
> *The main source of a compound Transition* T *is a direct substate of LCA[*T*].*
> *The main target of* T *is a substate containing all explicit targets of* T*.*

For this, note that the *main source* of T must be contained within the *State* that contains the whole compound *Transition* T, otherwise the *LCA*[T] may be pushed to a lower level. Since the *main target* must be contained in the *LCA*[T] and cannot be identical to *LCA*[T], it is a *State* inside *LCA*[T] that contains all explicit *target*s.

Within its *LCA*, an enabled and firing compound *Transition* proceeds to the execution of the following sequence:

1. Exiting of the *main source*.
2. The *ActionSequence* of the *effect* association of the *Transition* is performed.

3. Entering of the *main target*.

These steps simply extend the transition execution sequence for simple *Transition*s.

A *StateMachine* is the description of the possible actions and responses that may occur in the behavior of an *Object* or *Interaction*. Abstractly, it is a graph of possible *State*s and *PseudoState*s of the considered *Object* or *by Interaction*, that are connected by *Transition*s that are specified by execution conditions. The conditions are given by *Guard*s attached to the edges of the graph.

> **StateMachine** is a direct *ModelElement* subclass. A *StateMachine* is composed by a number of *StateVertex* instances and corresponding *Transition*s. The *ModelElement* whose behavior is specified, weakly owns the *StateMachine* via the *context* association. This association in not necessary one to one.

The *context* of a *StateMachine* is restricted to be a *Classifier* or a *BehavioralFeature*. A natural multi-scaling occurs for *StateMachine*s. Using the definition of *CompositeState*s, a *StateMachine* strongly owns one and only one highest-level *CompositeState*, designated by the *top* association. The *top CompositeState* cannot be the *source* or *target* of any *Transition*, neither can it contain any history vertices. All further relevant *State*s are transitively owned by the *StateMachine* through *CompositeState*s. This holds for *Transition*s too, the strongly owned *Transition*s are given by the *transitions* association, whereas all others are contained in *substates* as *internalTransition*s. The static semantics of the *StateMachine* metaclass are shown in fig. 5.16.

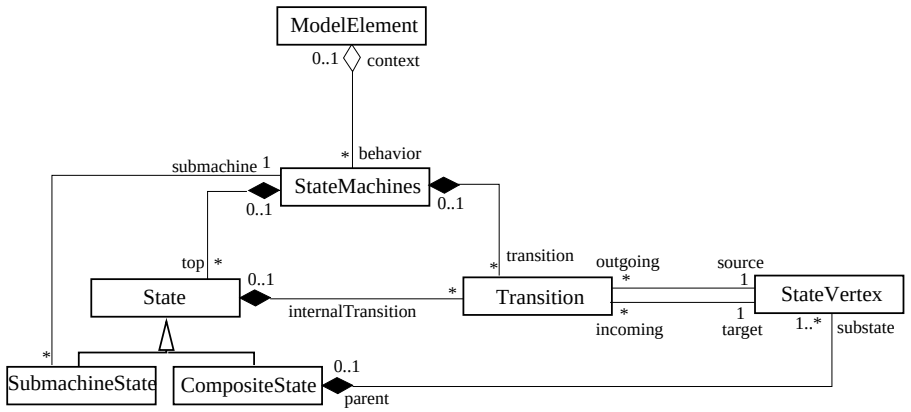


Figure 5.16: The static semantics of a *StateMachine*.

In the special case, when a *StateMachine* describes the dynamics of a *BehavioralFeature*, the only possible *CallEvent* trigger occurs in the initial *Transition*.

The functionality of a *StateMachine* consists of the dynamics of its including *State*s and *Transition*s. A *StateMachine* terminates, when an included stable *State* is active. Stable

*State*s are *State*s, whose *outgoing Transition*s are not enabled, that is, all corresponding *Guard*s are not fulfilled. As long as no active *State* in a *StateMachine* is stable, it represents a unstable, transient configuration and causes further steps to reach a stable *State* (although it may be impossible to finally reach one).

5.5.5 CONFLICTING TRANSITIONS

In general, due to *fork*s and *branch*es, in a *StateMachine* there may be more than one active *State*, simultaneously enabling more than one *Transition*. Further, since the *Guard*s of the *Transition*s with identical *source* are not restricted to provide an orthogonal decomposition of an expression, several enabled *Transition*s with identical *source* may occur. The latter situation is called a conflict situation. In this case it is not assured that the *StateMachine* evolves into a well-defined configuration. *Transition*s, which possess at least one common *source* with a conflict situation, are called conflicting *Transition*s. For example, since *internalTransition*s do not change their *State*, they may only conflict those *Transition*s, that cause the exit from the *State*.

To resolve a conflict situation, when more than one *Transition*s departing from the same *source* are enabled, an explicit *transition selection function*, specifying their firing order is defined. However, a conflict situation may be resolved by introducing priorities for the *Transition*s. In this sense, respecting the substate-hierarchy of the *source*s one gets a partial ordering of the *Transition*s by the following definition.

> A *Transition* T_1 has higher priority than T_2, if and only if *source*$[T_1]$ is a proper substate of *source*$[T_2]$.

In a compound *Transition* originating from a join, the priority is given by the priority of the lowest included *source*. Since restricted to hierarchical *source*s, this ordering cannot be extended to include all *Transition*s.

Once the *transition selection function* is defined explicitly, or implicitly by priorities, the set of firing *Transition*s is given by the conditions:

- All included *Transition* are enabled.

- No conflicting *Transition* are included (non-trivial).

- All enabled *Transition*s not included in the set must have priorities lower than the lowest one in the set.

The definition of priorities used in the UML is reversed compared with the one used in classical statecharts. This is reasonable, since substates are specializations of their containing ones, and therefore may override them.

5.5.6 ACTIVITY MODELS

One useful extension of *StateMachine*s are *ActivityModel*s. In essence, they correspond to *StateMachine*s and are used to describe processes involving a collection of *Classifier*s as finite state-transition systems. In *ActivityModel*s, *Transition*s are triggered by completion of previous *State*s, by the availability of an *Object* with a certain *State*, by the occurance of a *Signal*, or by satisfaction of some condition. The semantics of *ActivityModel*s are given by the semantics of a *StateMachine*, that is thought to be applied to a collection of *Classifier*s, rather that to a single one, and are only slightly modified for this purpose. As is the case for *StateMachine*s one may proceed in two different ways, first to focus on the *Object*s, connecting them by activities, or second to focus on the activities themselves. The former choice is known as Gantt charts, the latter as Pert charts. In the UML this distinction corresponds to interaction diagrams of *Collaboration*s, or to activity diagrams of *ActivityModel*s, respectively. The metaclasses used in the specification of a *ActivityModel* are shown in fig. 5.15.

Most *State*s in an *ActivityModel* are *ActionState*s. They are used to represent the invocation of an *Operation* in a *Classifier* as an atomic action and build the analogon to *SimpleState*s in *StateMachine*s.

> **ActionState** is a direct *SimpleState* subclass. Its *entry* association, inherited from *State*, specifies a non-empty *ActionSequence* performed, when a *Transition* enters the *ActionState*. The unique outgoing *Transition* is triggered by completion of the *entry* actions.

Other *Transition*s do not possess *ActionSequence*s. Another *SimpleState* is the *ObjectFlowState*, which designates an *Object* to be available when a certain *State* is active. This is used for example when an *Object* in a certain *State* is required as an input for an *ActionState*. Usually the *ActionState* execution changes the *State* of the *Object* and causes thereby a new *ObjectFlowState* describing the new situation.

> **ObjectFlowState** is a direct *SimpleState* subclass. The *typeState* association of an *ObjectFlowState* relates it to the *Classifier* and *State* of the current *Object*.

The *SubmachineState* metaclass has an analogon too, it is given by the *ActivityState*.

> **ActivityState** is a direct *SubmachineState* subclass.[7] An *ActivityState* describes a nested *ActivityModel*, whose semantics are given by substituting it hypothetically by the *StateMachine* specified by its *submachine* association.

To characterize the *Instance*s of a *Classifier* for a particular *State*, the *ClassifierInState* metaclass is used. *ClassifierInState*s are used for *ObjectFlowState*s.

> **ClassifierInState** is a direct *Classifier* subclass. The *type* and *inState* associations designate a *Classifier* and a valid *State* of this particular *Classifier*.

[7] In the figure on page 121 of the semantics specification [81] this is not shown correctly.

*PseudoState*s may be used in *ActivityModel*s with the same semantics as in *StateMachine*s. Moreover, to enable the detailed structuring of an *ActivityModel*, the *Partition* metaclass is introduced. Partitions group the *State*s in an *ActivityModel*, without changing the dynamic semantics of the model.

*ActionState*s, *ActivityState*s, and *ObjectFlowState*s are combined within the *ActivityModel* metaclass. An *ActivityModel* describes the processes in a system, that involve a collection of *Classifier*s, as control- and object-flows. These flows are presented using *StateMachine*s and related metaclasses. The semantics of the used metaclasses are not extended for this purpose.

> **ActivityModel** is a direct *StateMachine* subclass. The *context* association, inherited from the *StateMachine* metaclass, attaches the *ActivityModel* to a *Package*, a *Classifier*, or a *BehavioralFeature*, whose behavior it describes.

The *partition* association of an *ActivityModel* connects it to a *Partition* that divides its *State*s according to certain criteria, which may not have an interpretation in the UML. *Partition*s are used to structure processes in high-level system modeling, for example in business process modeling.

In principle, *ActivityModel*s correspond to data-flow models for general process modeling. However, it is important to note, that there are some essential differences. In *ActivityModel*s the *Operation*s in *ActionState*s always originate from *Class*es of types, *ObjectFlowState*s are well-defined and explicit *State*s of the UML-model, and the content of an *ObjectFlowState* is a formalized expression in the UML. These advantages of *ActivityModel* clearly motivate their usage in object-oriented models.

5.5.7 VIEW FROM THE OUTSIDE: USE CASES

Usually, systems that are modeled by the UML are not considered as isolated entities. A modeled system is assumed to possess a boundary separating it from its environment. Moreover, the environment of the system is thought to consist of a collection of elements, which interact with the system in a specific way. These interactions build the only relation between system and environment. External elements may be instances of classes that describe 'users' of the system, or other systems. They are called *Actor*s in the UML. The interaction of actors with the system causes the system to perform some internal behavior, which is realized as a collaboration of internal elements. The performance of the behavior activated internally cause the response of the system.

A central idea in the UML is the separation of internal and external views. This means, that the internal behavior is not part of an external view. For the scenario described above this implies, that the concrete internal collaboration that produces the system's response is not visible from the outside, that is, not visible for the *Actor*. Instead, the sequence of internal actions is represented by a *UseCase*, to which the interacting actor is directly related.

In this sense, viewed from its surroundings a system looks like a black-box providing some specific interfaces for interaction.

Since a system may be contained in another one as its subsystem, the definition of internal and external view is a relative one. The actors denoting the environment of a subsystem may be internal elements of the containing system. Similarly, use cases of the containing system may be realized by collaborations incorporating use cases of the subsystem. This motivates a generic scaling of use cases in a model describing the use cases. From the point of view of ToCS this can be considered as a composition of objects. The relations of external objects are defined as relations to the composite object, the internal structure is visible from the outside only by interfaces.

The Use Case package

The Use Case package is a relatively small package in the metamodel, that provides the metaclasses *Actor*, *UseCase* and *UseCaseInstance*, as well as the relations that may connect them. The package depends on the Foundation package and on the Common Behavior package.

An *Actor* defines a set of roles for the interaction with an entity. An entity in this section is always meant to be a *System*, a *Subsystem* or a *Class*.

> **Actor** is a direct *Classifier* subclass. An *Actor* may have *Association*s to *UseCase*s and is owned by a *Namespace*.

*Actor*s can only have model-specific *Association*s to *UseCase*s. These *Association*s must be binary, for each connected *UseCase* the Actor must have one role. To communicate with other elements, an *Actor* may offer a set of *Interface*s, whose *Operation*s must match with *Operation*s in the *Actor*. Being a *GeneralizableElement* subclass, *Actor* instances may replace instances of ancestor *Actor*s. This may be used to refine the environment of an entity. An *Actor* can not contain any other *Classifier*.

*Actor*s in the environment of an entity are related to the entity by *UseCase*s. A *UseCase* captures a part of the behavior of the entity as a sequence of internal actions, without specifying its implementation. *UseCase*s cannot contain *Classifier*s.

> **UseCase** is a direct subclass of *Classifier*. A *UseCase* contains *Attribute*s and *Operation*s for the specification of its behavior.

To specify the realization of the *UseCase*'s behavior, *Collaboration*s of *Instance*s of the *ModelElement* containing the *UseCase* are used. A *Collaboration* defines how the specific *Instance*s interact to produce the specific behavior. The performance of the sequence of actions of a *UseCase* is called a *UseCaseInstance*. With respect to the type/instance dichotomy *UseCaseInstance*s are the instance analogon to *UseCase*s.

UseCaseInstance is a direct *Instance* subclass. A *UseCaseInstance* is related to a *UseCase* as its implementation.

The *Method*s performed by a *UseCaseInstance* are atomic transactions, that is, they cannot be interrupted by other *UseCaseInstance*s in the run-time-system. Figure 5.17 shows the abstract syntax and some of the static semantics of the Use Case package constructs.

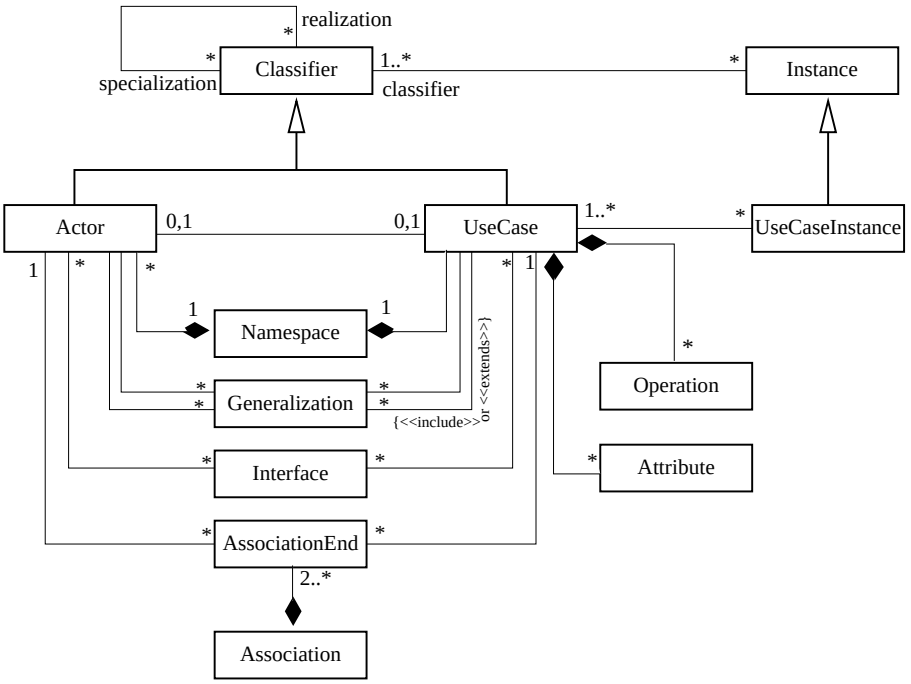


Figure 5.17: The abstract syntax and static semantics of the *Actor*, *UseCase* and *UseCaseInstance* metaclasses, contained in the Use Case package.

All *Association*s of *UseCase*s must be binary. In particular, *UseCase*s are related to *Actor*s and to other *UseCase*s. However, *UseCase*s of the same entity can not be related by *Association*s, since every *UseCase* is supposed to present a complete functionality of the system at the given level of abstraction. *Association*s to *UseCase*s are allowed only, when the *UseCase* are contained in different packages, although the packages may be contained in each other. We will come back to this point later. *UseCase*s in the same entity may be related only by stereotyped *Generalization*s. Two stereotypes are used for this purpose: `<<include>>` and `<<extend>>`. The `<<include>>` stereotype defines the delegation of interactions of a base *UseCase* to other *UseCase*s, that is, their explicit incorporation.[8] On the other hand, the `<<extend>>` relation defines an extension of the *Operation*s

[8]This stereotype was called '*uses*' in the UML version 1.1 (September 1997). The release 1.3 (1998) changed notion [89].

sequence in the base *UseCase* by the *Operation*s in another *UseCase*. The location where an `<<extend>>` relation may arise is given in the *UseCase* as an attribute with the name *extensionPoint*, which is a list of strings. The `<<extend>>` relation specifies the condition that must be fulfilled for the *UseCase* to be extended.

The fact, that *UseCase*s may have *Association*s to *UseCase*s of contained *ModelElement*s leads to a generic scaling mechanism, when the system in specified via successive refinements of external views. For this suppose a hierarchy of *Subsystem*s is given. The environment of a *Subsystem* is the *ModelElement* that contains the *Subsystem*. The specification of the behavior offered by the system may then be given by *UseCase*s at all levels. In this case, both, the *Actor*s and the *UseCase*s of a containing system are *Actor*s of the contained subsystem. The internal structure of the superordinate *UseCase* is then given by collaborations of its subordinates *UseCase*s. This means, that the interactions of the superordinate are transmitted to its subordinates. The performance of the superordinate is the performance of the collaboration of its subordinates. The response of the subordinate collaboration is transmitted to the superordinate *UseCase*. Here, *Collaboration*s are used both, to refine a *UseCase*, and to specify its realization. In refining *UseCase*s, *Collaboration*s of *Class*es may be used, whereas to refine *Class*es, *Operation*s contained in the *Class*es may collaborate. This consideration leads to a uniform specification of the whole system. Moreover, modifications on a certain scale are traceable to its subordinates as well as to its superordinate. Due to the elementary character of the constituents of such considerations one may expect easy simulations of their structure at this level.

Finally, in developing information systems, requirements on a coarse scale may be stated in form of use cases guiding the architecture of the system and providing a measure for the stage of implementation by internal processes. Models described exclusively by use cases produce stereotyped *Model*s, called `<<useCaseModel>>`. Modeling approaches that specify the functionality of a system first are called use-case-driven. The implementation of the concrete behavior is considered at advanced stages.

5.6 SUMMARY

In this chapter the metamodel of the *Unified Modeling Language* (UML) was analyzed and some corrections to the original specification were made. The UML is a general-purpose modeling language for the specification, construction and representation of object models and of behavioral models, where the former describes the structure and the latter the dynamical behavior of objects in a system.

After explaining the object-oriented metamodeling approach, the UML-metamodel, defining structure and semantics of UML-models, was investigated. A UML-model is based on the concepts and constructs defined in the UML-metamodel. The UML-metamodel obeys itself the object-orientation paradigm, that is, its metaobjects are structured in form of a

generalization-hierarchy and related to each other by well-defined relationships. All elements of a concrete UML-model are assumed to be instances of the metaclasses of the metamodel. The general behavior of the model elements is assumed to be taken over from the predefined behavior properties of the metamodel. To be able to apply the UML approach in semantically different contexts, integrated extension mechanisms are provided by the metamodel, that allow to extend the semantics of existing metaclasses in a well-defined fashion. It is explained how the instantiation of a model is defined in the metamodel. The model elements introduced so far may interact in collaborations, in order to provide complex behavior. The internal dynamics of model elements are specified using finite state machines. FSMs are extended to describe processes. Finally, it is explained how UML-models interact with their environment.

Chapter 6

DISCUSSION

This chapter presents a discussion of the inherent *multiscaling properties* in the UML. They are used as a canonical method to reduce the system complexity. The second section relates all modeling approaches introduced in the previous chapters. For this, a recently proposed systematic classification and integration scheme is explained and applied to the modeling techniques of the last chapters. A comparison according to this scheme enables the embedding of ToCS into the wide area of system specification techniques.

6.1 IDENTIFICATION OF MULTISCALING PROPERTIES

In the UML the increasing complexity of a system is treated with the help of multiscaling mechanisms, which are based on the definition of composite structures of model elements on higher levels of abstraction. As argued in chapter 1, composite objects are the basic concept for the definition of a multiscale treatment of a complex system. Usually, different scales in the UML are not distinguished by their time or space scale, but due to the level of abstraction of their containing objects. One may distinguish two separate sectors for multiscale mechanisms, structural multiscaling and behavioral multiscaling.

- **Structural Multiscaling**
  Structural multiscaling is realized in the UML by constructing new structural elements out of more elementary ones. In this sense, *Attribute*s are constructed from *DataType*s, while they are contained themselves in *Classifier*s to describe the states of their instances. *Class*es may be composed to build composite *Class*es. Further, all kinds of *Classifier*s are combined into *Subsystem*s, which provide the main constituents for *Model*s and *System*s.

- **Behavioral Multiscaling**
  Here, the basic mechanism is given by the *Collaboration* metaclass. In this sense, the behavior of *Operation* are realized by collaborations of *Method*s, while they are used

as constituents for collaborations which realize *Class*es. However, collaborations of *Classifier*s are used to define the realization of *UseCase*s, whereas, being *Classifier*s, *UseCase*s may collaborate to specify superordinate *UseCase*s.

Of course, the concrete analysis of these scaling mechanisms requires the knowledge of the specific model, in which they occur. However, based on the metamodel, it is possible to sketch the basic mechanisms that cause scales in almost any model. To clarify how the structure of a UML-model is based on these mechanisms, some mechanisms are described in more detail.

Packages and Subsystems

The most general scaling mechanism is given by the *Package* metaclass. However, to get an operational hierarchy of model elements, it is important to structure the model into elements, that provide both, a container for other elements, as well as a specification of its dynamical behavior. Therefore, a combination of the *Package* and *Classifier* metaclasses is needed. This is provided by the *Subsystem* metaclass. Scaled models should therefore offer a layered system structure, where every layer contains *Classifier*s. This means, that subsystems may again be decomposed into several (nearly independent) subsystems.

Dependencies

The association analogon to the grouping of elements into subsystems, is provided by the *Dependency* metaclass. If a hierarchy of *Subsystem*s is given, the *Dependency* relations between elements inside to elements outside the *Subsystem* are consolidated into new composite *Dependency* relations between *Subsystem*s. The weak ownership relations to the constituents are given by the *subDependency* association. Due to this mechanism, it is possible to analyze the coarse structure of a model in terms of *Subsystem*s and their *Dependency* relations.

Use Case Models

Suppose the structure of a system is given in form of a *Subsystem* hierarchy. Then, the external views to the several *Subsystem*s may be given by a *useCaseModel* that describes how *Actor*s outside a *Subsystem* are connected to *UseCase*s in the *Subsystem*. From an abstract point of view, this constitutes a layered system. The layers are *Subsystem*s, the containing objects are *UseCase*s. The inter-layer-links are the binary associations between *Actor*s and *UseCase*s. Note, that the *UseCase*s may be composite ones, composed from the ones in a containing *Subsystem*.

Collaborations

The standard usage for collaborations in the UML is to define *Operation*s by collaborat-

ing *Method*s. In addition, *Class*es are realized by their collaborating *Operation*s. Then, the behavior of the *Class* is given by the union of all subordinate *Collaboration*s of *Method*s. However, in the specific case, that two *Collaboration*s of *Method*s arise, they should be fused into a larger, composite *Collaboration*.
Another example is given by the fusion of collaborating *UseCase*s. This arises, when a superordinate *UseCase* is realized by a *Collaboration* of subordinate *UseCase*s. In this case, the scales are determined by a *useCaseModel*, that is, by the definition of a subsystem hierarchy in terms of *UseCase*s, which provide the external views. The *UseCase*s of a specific view may be refined by relations to the subordinate *UseCase*s. For the realizing *Collaboration*s this means, that they are combined to provide the realization of the superordinate *UseCase* on the next scale. The composition of *Collaboration* was explained in the last chapter. It relies on the fact, that *Collaboration*s do not contain *Classifier*s, but *ClassifierRole*s. From an abstract point of view, the composition mechanism may be considered as an enzymatic action that corresponds to motion with trace. Two collaborations are composed by fusion of attributes and operations of those *ClassifierRole*s, which are related to the same *Classifier* by their *base* association. They designated to identification by a path, consisting of the two *base* associations to the same *Classifier*.

Of course, there are further mechanisms, similar to the described. For example, *State*s are composed to *CompositeState*s in standardized fashion. However, most scaling-mechanisms rely on the usage of *Collaboration*s.

6.2 CLASSIFICATION AND COMPARISON

There are some main reasons to provide a systematic scheme for the comparison of different modeling techniques. First, the structure of many interesting systems may be too many-sided, to be appropriately modeled by a single approach. In this case, one has to simultaneously use more than one modeling technique, to encompass all relevant aspects. Since the used techniques should model the same original system, the corresponding models will not be independent of each other. To provide a complete conceptual description of the original system, it is important to integrate the various models into a single conceptual one, or at least, to precisely clarify their interrelations. This is very hard to achieve, since the used techniques may have rather different starting points, and may provide different views of the modeled system, concerning with different aspects. Therefore, it is necessary, to structure the used techniques according to some general scheme, which enables the comparison of the 'right' levels. In addition, a general scheme, may help clarifying in principle, if the mixing of levels, belonging to different models, is possible.

The second main reason is, that a systematic comparison supports the decision process, that is, it helps to decide, which modeling technique is used best for specific aspects, by clarifying the advantages and disadvantages of the approaches.

In addition, the usage of an accepted comparison scheme, helps embedding the ToCS-based approach into the area of specification techniques.

Coarse classification

The general scheme used here for the comparison of the modeling approaches, that have been presented in the previous chapters, is based on a recently proposed classification and integration paradigm by Ehrig and Orejas [101]. Before introducing this scheme, the presented approaches are classified coarsely into two main classes, the process-driven approaches, and the data type-driven approaches. Process-driven approaches concentrate first on the processes occurring in systems, that is, on their behavior. The used data types are either primitive, or specified implicitly by the requirements of the processes (reactive states). Conversely, by using formal, or more general specification techniques, in data-type-driven approaches the used data types must be defined first. The processes in the system, describing its behavior are defined as transformations of data states builded out of the specified data types. The modeling techniques of the previous chapters may be categorized with respect to this classification as follows:

- **Process-driven approaches**
  Premium examples for process-driven approaches are P/T-nets. The processes are encoded in the structure of the net, while the used data types are primitive tokens. Similarly, state-transition systems like statecharts, or finite state machines concentrate on the reaction of the system, the states are given by reactive states, defined by the behavioral requirements. In graph transformations, too, the modeled processes are encoded in the graph transformation system first.

- **Data-type-driven approaches**
  Premium examples for data-type driven approaches are mathematical modeling methods. There, the used entities must be defined completely, before starting any kind of modeling. Object-orientation is a data-type-driven approach, too, since modeling a system according to the fundamental concepts of object-orientation requires the definition of the used classes first. In this sense, the UML is a data-type-driven approach, since it incorporates the object-orientation paradigm.

The ToCS-based approach is considered as a hybrid approach. The definition of a system by objects and arrows suggests its classification as data type-driven, since basic data types may be defined as systems themselves. However, the structure of a system may define the processes in the system, too, as illustrated for example in P/T-systems. Therefore, the classification above is not thought as a complete classification, but rather as a coarse distinction of basic differences of modeling approaches.

Classification and integration paradigm

The paradigm used in the following to compare the presented approaches, extends the concept of **Dynamic Abstract Data Types** (DADT) [102]. DADTs generalize the concept of

abstract data types (ADT) (see chapter 4) to incorporate dynamical behavior. Like abstract data types, DADTs support the modularity principle by encapsulating the defining data and the operations that are allowed to manipulate the data. DADTs are defined constructively. Their structure and dynamical behavior are specified in four layers as follows:

- DADT-L1: **Value Type Specification**
  The used values are specified by formal algebraic methods with fixed semantics for the value domains and the operations that may operate on them.

- DADT-L2: **Instant Structure Specification**
  The elements of DADT-L1 are extended by algebraic specifications allowing constraints to be imposed on them , for example data tuples restricted to subsets of the product of the domains. Value algebras on this stage represent the states of the system, which are also called instant structures.

- DADT-L3: **Dynamic Operation Specification**
  Contains operations, which specify the possible state transformations. The layer DADT-L3 is *not restricted to algebraic specifications*, other techniques like finite state machines, or P/T-nets and higher-level Petri-nets, or graph transformations may be used.

- DADT-L4: **Higher Level Dynamics Specification**
  Here, the allowed large scale modifications, like composition and communication of constructs from DADT-L1, DADT-L2 and DADT-L3 are contained.

DADTs are algebraic specified types and states defined in DADT-L1 and DADT-L2, which are extended in the layers DADT-L3 and DADT-L4 by allowing *non-algebraic specifications*. DADTs arise in many contexts. Of course, similar concepts have been used in various object-oriented approaches long ago [103]. Examples for DADTs are given in [104]. However, many established modeling approaches cannot be brought into a form compatible with the DADT paradigm. One reason is, that even in layers DADT-L1 and DADT-L2, *it is too restrictive to require algebraic specifications*. Another crucial point is, that the layer DADT-L3 does not provide an appropriate treatment of processes. As mentioned, in process-driven approaches, the specification of processes is the starting point in the model, while the used data types are defined only implicitly as reactive states, like the states in a state machine. *A separate process layer* would therefore help to clarify the relations between states, defined by the basic data types, and reactive states, defined implicitly by the behavior of the system. Reactive states should therefore play a twofold role, being abstractions of basic data states, and defining the states during the lifetime of a system. Finally, there is a need to incorporate any type of horizontal structuring mechanism in a separate layer.

One realization of the requirements stated above is provided by the recent four layer *Integration Paradigm* of Ehrig and Orejas [101]. It classifies a modeling approach due to four conceptual layers, shown in fig. 6.1.
The four layers in fig. 6.1 are defined as follows:

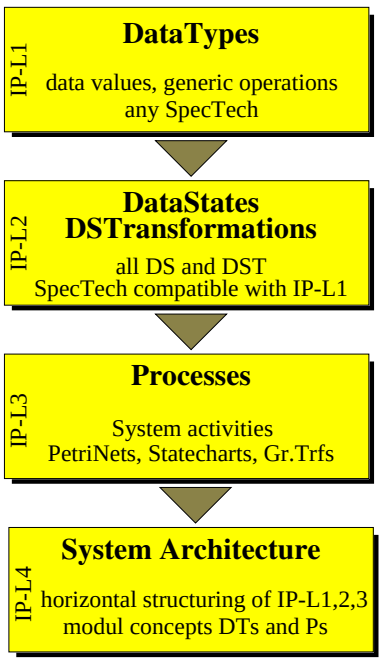


Figure 6.1: Four layer integration paradigm for classification of specification techniques according to Ehrig and Orejas [101].

- **IP-L1 : Data Types**
  As in DADT-L1, here the basic data types used in the modeling approach are contained. This means, that all elementary values and operations/functions that act exclusively on primitive values are specified in this layer. The specification need not to be algebraic, and extends therefore DADT-L1 essentially.

- **IP-L2 : Data States & Transformations**
  This layer provides the collection of possible states for the system, although they may not be realizable by the system during its lifetime. The states are defined based on the data types of IP-L1. The transformations that apply on the states and represent corresponding state-changes are included in this layer, too. Abstractly, the states and transformations in IP-L2 are considered to provide a state-transition-system. The dynamics of the system is then a subsystem within this system.

- **IP-L3 : Processes**
  Here the processes of the system are specified. Processes represent the dynamics of a system as a whole. Considering the state-transition-system of IP-L2, the processes in IP-L3 should provide an abstraction of this system, or at least of parts of it, for example by combining elementary transitions to form composite ones.

- **IP-L4 : System Architecture**
  Any horizontal structuring method for the model is contained in IP-L4. In general, this requires concepts for the union and decomposition of model elements from or into sub-elements. Moreover, a module concept for the modularization of the system should enable the definition of components of the system, with precise structure, behavior, interrelations, interfaces and further semantics. Right now, this layer is supported only by few modeling approaches.

Note, that the Integration paradigm introduced above, has a hierarchical structure, in the sense, that a following layer is based on its predecessor. Using this classification scheme, the modeling approaches of the previous chapters are described as follows:

- **P/T-nets**
  In P/T-nets, the layer IP-L1 contains only trivial data types in form of atomic tokens located in places. The data states in IP-L2 are given by markings of the net, the corresponding state transformations are given by the firing mechanism of transitions. The essential information of a P/T-net is encoded in the structure of the net, which describes the modeled processes. Therefore, the layer IP-L3 contains the net processes. Since there are no generic union and fusion concepts in classical P/T-nets, the layer IP-L4 is not supported. As mentioned in chapter 3, using higher-level nets, one may overcome this inability.

- **Graph Transformations**
  In (low-level) graph transformations, the layer IP-L1 provides the naming used in the graph, and contains therefore an label alphabet as simple data type. The state are given by the graph itself, while the graph transformations defined for the graph build the state transformations of layer IP-L2. Coupled graph transformations and graph processes provide the elements in layer IP-L3. Right now, there are no established horizontal structuring, or composition concepts for graph transformation systems.

- **Statecharts**
  The statechart technique is chosen as an example for a state machine approach, since it is used in the UML, too. In a statechart the layer IP-L1 contains the basic data types that are used to define the states in IP-L2 by specifying their values (variable binding). Being a process-driven approach, the processes in IP-L3 are given by the statecharts themselves. IP-L4 supported by using activity charts based on the statechart technique to horizontally structure the model.

- **Object-Orientation**
  In object-orientation, the layer IP-L1 contains basic data types that may be used as attributes in classes and objects. These are for example integer, or strings. The layer IP-L2 contains the states of the objects defined by values for the containing attributes as its data states. Further, basic data transformations are defined to be the operations defined in the classes. The layers IP-L3 and IP-L4 are not properly supported, although for IP-L3, one may use the operations in the objects to define more complex processes.

- **UML**
  Since the UML supports object-orientation, the layers IP-L1 and IP-L2 are given by the instances of *DataType*, and *State*s of the *Object*s together with the corresponding *Operation*s and *Method*s. For processes, object state machines based on the statechart technique are used. The *Package* metaclass provides a horizontal structuring concept contained in IP-L4. Further, large scale models, for example use case models, or activity models may be used to structure the model, too.

- **ToCS/SyCL**
  The ToCS approach and its implementation in SyCL (see appendix A) support the layer IP-L1 in form of the constituents of a system, that is, by objects and arrows as simple data types. However, since objects may possess internal structure and arrows may be attributed by values, one may use almost any data type in system. This allows the modeling of a large class of systems. In IP-L2 the states are given by the system itself. As shown for P/T-systems, the state may be given by a part of the actual system. The state transformations are given by all allowed actions of fundamental enzymes. In particular, every enzyme implementing a universal dynamics law, should be applicable. Processes in layer IP-L3 are given by sequences of systems evolving due to the actions of composite enzymes on the system.

  The horizontal structuring mechanism supporting layer IP-L4 in ToCS/SyCL, is the notion of composite objects, causing the existence of several scales in a model. In addition, since the basic mechanism for horizontal structuring is the fusion of subelements, one may use composite links to define the fusion of systems in terms of ToCS.[1] In this sense, the fusion of two subsystems A and B without non-trivial automorphisms into a larger one C is performed by connecting them by a composite link. A *composite link* that connects the subsystems A and B is a system L, whose boundary ∂L consists of two disjoint subsystems ∂L_1 and ∂L_2, where the targets in ∂L_1 are isomorphic to the objects of A and the ones in ∂L_2 to the objects in B. In this sense, one may think that the composite link L sticks together the subsystems A and B by incorporating their arrows. However, if the boundary of L does not fit exactly the subsystems A and B one may expect a kind of interface to realize the connection. In this sense, by considering the subsystems as composite objects, the fusion of subsystems provides a kind of multiscaling mechanism.

The comparison is represented on the following table. It turns out, that the ToCS/SyCL approach provides both, a conceptual framework, as well as comprehensive inherent mechanisms.

[1] G. Mack, private communication.

Model. Appr.	IP-L1	IP-L2	IP-L3	IP-L4
P/T-nets	tokens	markings, firings	net processes	
Graph Transf.s	alphabet	graphs graph transf.s	graph processes	
Statecharts	basic DTs	data states by variable binding	statecharts	(activity dgr.)
Object-Orient.	basic DTs	classes	(object behavior)	
UML	basic DTs	classes	statemachines activity dgr.	(packages)
ToCS/SyCL	arrows objects	systems, fundamental enzymes	composite enzymes	composite objects and links

6.3 SUMMARY

The identification of the fundamental multiscaling mechanisms of the UML and a comparison of all presented modeling approaches based on a recently proposed classification scheme were given in this chapter.

Chapter 7

CONCLUSIONS AND OUTLOOK

Let us briefly summarize the main points obtained in the thesis:

- A general analytical setting was presented for the description and analysis of structural and dynamical properties of complex systems. It was explained what the typical dynamics may be.
- A precise definition for composite objects was given. This was motivated by the definition of boundaries for categories. It was shown that composite objects occur naturally, if bidirectionally connected, unfrustrated subsystems are present. Some remarks concerning the handling of composite objects were made.
- Some common modeling techniques for information systems were embedded into the ToCS-framework. In particular, the structure and dynamics of P/T-nets with firing transitions were formulated as ToCS-systems with enzymatic actions. The equivalence of both formulations was shown, and it was explained why a ToCS-based version may overcome most inabilities of the classical formulation.
- The ToCS-based formulation of classical electrodynamics was carried out. It extends the results in [31] to the case of present electric sources and currents. Examples were given.
- The massless Dirac-equation was translated into a ToCS-based form. The somewhat technical construction might be used in other scenarios, too.
- It was argued why information systems that obey the object-orientation paradigm enable a ToCS-based treatment. The conceptual formulation of object-orientation was given a ToCS-based formulation.
- The metamodel structure of the UML was analyzed in detail. It was shown that the metamodel elements are structured inherently by common mechanisms, like the type/instance dichotomy, or the integrated extension mechanisms.

- The UML-metamodel was analyzed to explain how the specified entities are used as templates to construct models that obey the object-orientation paradigm. The explained concepts include the modeling of the dynamics of a system.
- Some corrections to confusions in the original specification of the UML-metamodel were given.
- The main complexity reduction mechanisms of the UML were identified as inherent multiscaling properties.
- Using a recently proposed classification paradigm the ToCS-based approach was embedded into the field of formal system specification. All presented modeling were compared by using the classification paradigm.

Of course, there are many points which remain to be elaborated. In general, it is important to proceed to an enzymatic construction of many advanced ToCS-concepts. For example it is interesting to analyze to what extend structures in a system may be recognized by the local action of enzymes. This is necessary, when composite objects should arise automatically. Although this is possible in special cases, it is interesting how far one may go in a general setting.

In addition, the enzymatic construction of functors is an important issue. In particular, the construction of representations is related to the recognition of composite objects, but in general, one is interested in analyzing for example the enzymatic construction of isomorphic systems.

The investigation of views for general systems is another important point. This includes the analysis of their consistency under dynamical laws. Further, it is important to reverse direction and ask, if it is possible to reconstruct a system, once a collection of views is given, that is, if it is possible to integrate the views to a single system, whose model the views constitute. Related to this point, one may proceed to the integration of different modeling approaches by formulating them in ToCS-based versions.

Further, it would be extremely important to provide an enzymatic formulation of the inherent mechanisms in the UML, in particular of all complexity reduction mechanisms by multiscaling. One may think of a UML-model optimization by ToCS-based processes. This requires a much more detailed analysis of the connections and a precise formulation of UML-mechanisms in the ToCS-framework.

Appendix A

ToCS/SyCL

The universality and algorithmic character of the structure and dynamics of the ToCS-based modeling approach, motivates the development of an implementation framework for the theory. An implementation should realize the concepts properly and enable the simulation of appropriately defined models. Such an implementation exists, based on an object-oriented class library, the *System Class Library* (SyCL). SyCL is to be used via a graphical user interface, as illustrated in [33]. This appendix provides a brief overview of general implementation aspects and of the basic concepts of SyCL. The sections on SyCL use ideas and work with Jan Würthner.

A.1 Beyond Analysis

Abstract modeling of a 'real' system produces a conceptual model that contains in more or less formalized fashion the essential constructs. However, the formulation of conceptual models of complex systems is often guided by the demand to enable implementations of these models as run-time-systems for various purposes, for example simulations of dynamical systems. How far a conceptual model supports its implementation, depends on the structure of the used modeling language. While some modeling languages are constructed with the scope to enable implementations, other approaches are far from enabling such issues. As shown below, ToCS supports implementation aspects properly, mainly due to its universality and to the finite-operation-character of the corresponding dynamics. In this sense, both, the conceptual analysis, and the implementational design is feasible in a uniform framework.

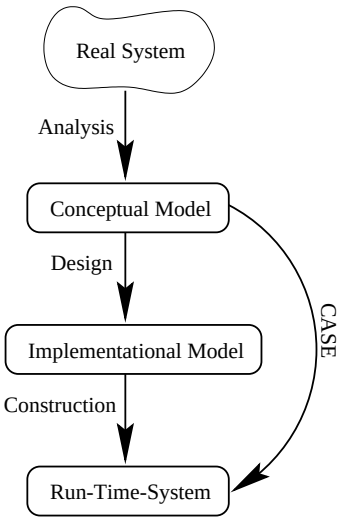
In general, in constructing a run-time-system one distinguishes three basic stages of model development [105]

- **Analysis**
  The analysis stage abstracts a 'real' system into a conceptual model, while trying to focus only on the essential aspects, from a logical point of view.

- **Design**
  The design stage translates a conceptual model into an implementation model, for example into the framework of a programming language.

- **Construction**
  In the construction stage an implementation model is realized as a run-time-system, like executing of an programming code on a computer.

Naively spoken, the analysis of a real system specifies what a model should do, whereas the design stage specifies how this is achieved. The construction stage states how the run-time-system is to be builded. While the implementation model depends on the used technology, the conceptual model resulting from the analysis is independent of the used design framework. If the analysis is performed based on appropriate theoretical methods, the development of CASE-tools (Computer Aided System Engineering) supporting both, design and construction issues, is a feasible task.



The use of CASE-tools is of special importance if when prototyping is used, that is, if first implementations should be produced, based upon a logical model of the modeled system. Prototyping is used mainly for reasons of consistency check of the conceptual model. CASE-tools are substantially involved in many object-oriented modeling approaches. The translation of object-oriented analysis concepts into object-oriented design constructs is an almost straightforward task. Rational's ROSE-tool is an example of a CASE-tool supporting the UML, whereas the SyCL tool described below, designs and constructs run-time-systems based on ToCS. The following section presents the most fundamental constructs of SyCL.

A.2 IMPLEMENTING TOCS: SYCL

In the modeling picture presented above, ToCS corresponds to the analysis stage, whereas SyCL is its corresponding CASE-tool, that enables the direct construction of a run-time-system based on a conceptual ToCS-model. Relying on an object-oriented class-library, it is possible to describe the modeling entities of SyCL as a generalization hierarchy. Fig. A.1 shows the most important classes that have to be used, when defining a model.

The idea of the implementation as a simulation tool is almost as old as the theory itself.

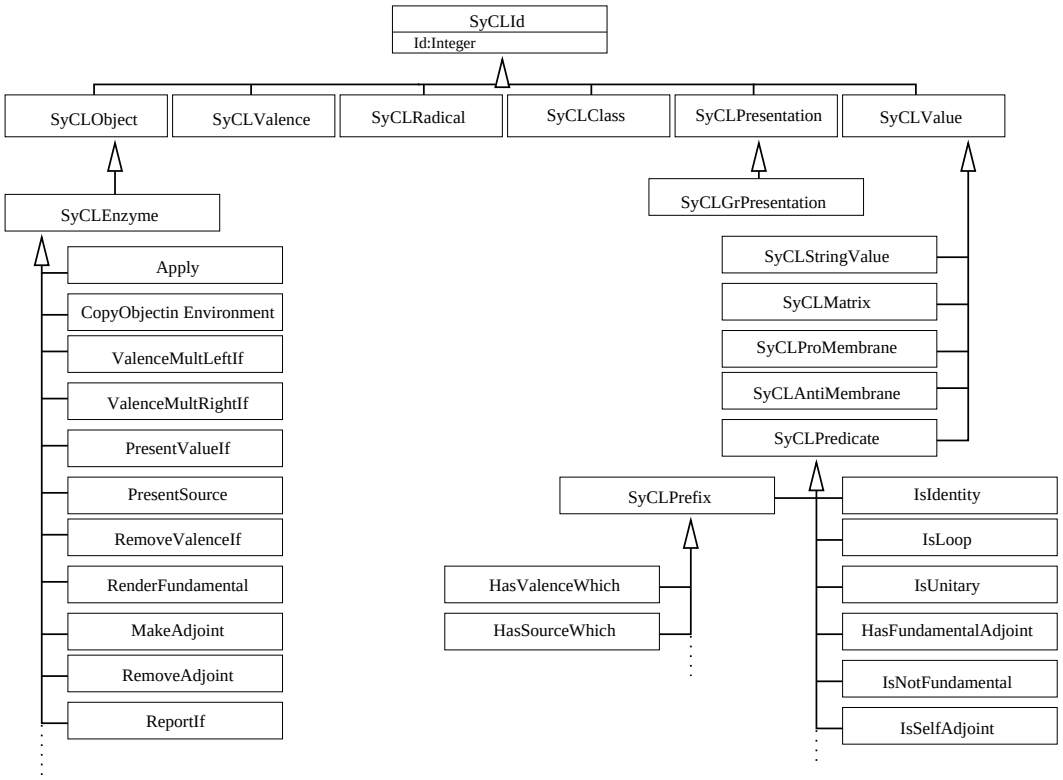


Figure A.1: The basic modeling classes of SyCL. The dots symbolize further subclasses. Classes concerning technical aspects are disregarded.

However, the view of the two has changed much during the development, and it took much work to clearly separate the theory specific from the technical aspects. [1] In this sense, the hierarchy of fig. A.1 focuses on the classes that manifest the correspondence to the ToCS approach. Technical aspects of the implementation are neglected.

The whole class hierarchy roots in the *SyCLId* class. This class contains one attribute *Id*, which is an integer. The *Id* attribute is used to identify every element of a system. From a technical point of view, the *Id* attribute enables one to write the system on disc. This is important to store initial conditions, or to run long-time simulations. Moreover, this attribute has turned out to be of great importance, when it is necessary to debug oncoming errors in a simulation.

The six direct subclasses of *SyCLId* shown in fig. A.1 implement the various concepts of ToCS. The *SyCLClass* class is mainly used to instantiate the descriptors of other classes.

[1] Yet, improvements implemented in the present may change some of the structure of the hierarchy in fig. A.1.

In this sense, there is a *SyCLObject*-descriptor, a *SyCLValence* descriptor, and so on. The *SyCLObject* class and the *SyCLValence* class are used to instantiate the objects and fundamental arrows of a system, respectively. The *SyCLEnzyme* subclass of *SyCLObject* is the class, that implements the universal dynamics concept of ToCS. Every *SyCLEnzyme* subclass is the implementation of one particular local, universal dynamics operation. There are a plethora of subclasses of *SyCLEnzyme*. Some of them are called fundamental, since it is expected to generate every other enzyme, by combining them into composite enzymes. The enzymes shown in the figure are fundamental ones, that combine to provide the system-copying *SplitFork* enzyme. This is shown in the next section. It is important to note, that every enzyme contains a naming attribute, called *genom*. Naively, this is a string expression of four characters, that are used to symbolize the specific enzymes. *SyCLRadical* enables one to combine a specific object and the fundamental arrows pointing towards this central object into a single entity.

The *SyCLValue* class is used to attach quantitative properties to objects and arrows. For example, a $m \times n$ matrix may be attached by associating a *SyCLMatrix* instance. The membrane classes are used to designate boundaries in a system, which encapsulate semantically distinguished regions, for example to restrict the application of an enzyme to a subsystem. The *SyCLPredicate* class enables one to attach expressions to enzymes, which are to be evaluated in the system by the carrying enzyme. Specialized predicates are *SyCLPrefix*. Object-orientation is used to assure the algebraic operations to be well-defined in any scenario.

A system is defined in SyCL as a *SyCLObject* instance, which contains internal structure. This means, that the instance owns a collection of *SyCLObject* and *SyCLValence* instances. In this sense, the SyCL implementation of a system, offers a proper realization of the multi-scaling concept of ToCS, based on composite objects.

The *SyCLPresentation* class concerns with presentation aspects of simulations. These aspects are out-placed enabling thereby the presentation of a modeled system in various frameworks. For example, the *SyCLGrPresentation* is a subclass offering a graphical presentation of a system. This structure reminds the treatment of representational aspects in the UML-metamodel. Nevertheless, it is important to note, that it was developed independently [33].

Finally, note that the hierarchy in fig. A.1 lies in the model-level. It is not a metamodel for building SyCL-models. A proper metamodel, in the sense described in chapter 5, should clarify the specific structure preferred in the SyCL implementation. A metamodel is under construction right now [33].

A.3 SYSTEM DEVELOPMENT WITH TOCS/SYCL

In order to extend other software by system-theoretic concepts, the SyCL-implementation initially started as a pure *C++* class-library. Now, the implementation has turned into an independent and self-contained software. A graphical user interface has been developed, to enable the run of simulations by encoding the data directly into SyCL rather then *C++*. In this sense, coding in SyCL means making use of generic ToCS-concepts. The crucial point here is not 'only' the ability to code directly in SyCL. Rather, it is the inability to evade it by using something else in combination with the SyCL classes [106]. This means, that a SyCL-simulation is a proper realization of system theoretical dynamics, based on local operations implemented by appropriate enzymes.

Although there is no formalized system design method, the concepts of the ToCS approach and their SyCL-implementation suggest a specific development method for the analysis and simulation of general complex systems. To carry out the simulation of a complex system one should go through the following steps:

- **Conceptual Analysis**
  Specification of the entities and relations in the system that is to be modeled. From this, the objects and arrows of the system are defined. Fundamental arrows and thereby the local structure of the system is identified. Specification of the composition law for arrows. Attaching of model-specific values, compatible with the composition law, to objects and links in the system.

- **Constraints**
  Encoding of the initial conditions and the general constraints into the system. This is realized by using the *language of thought*, that is, by pure structural requirements. The constraints should specify the isomorphism class of the system.

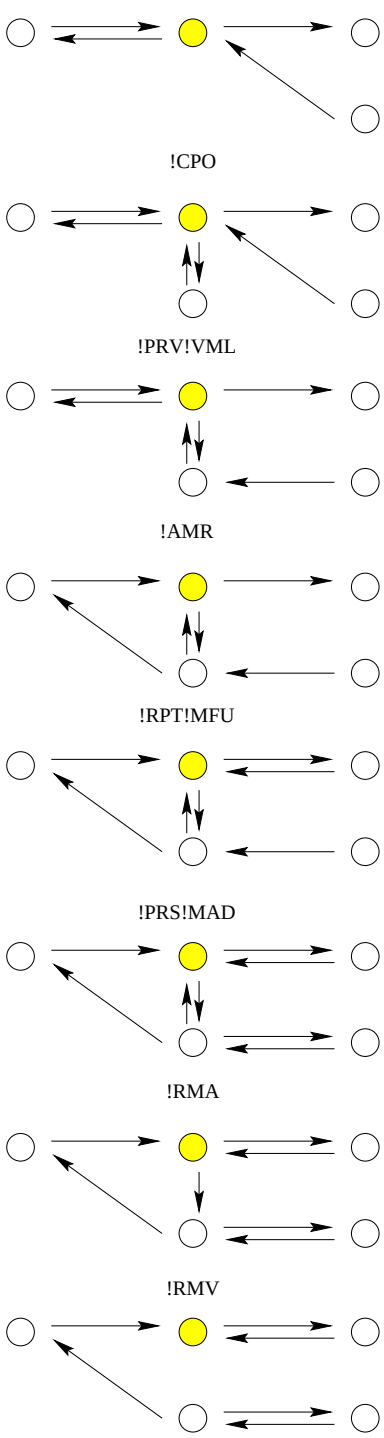
- **Dynamics**
  Specification of the dynamics occurring in the system. Formalization as enzymatic actions. Complex enzymes should be generated by using fundamental enzymes and composing them to composites.

- **Execution**
  Run the simulation. Extract data of the system, analyze composite structures and collaborations, etc.

All steps, necessary for the complete specification of the system should be encoded via the graphical user interface.

To illustrate the composition of enzymes in the dynamics stage, consider the *SplitFork* enzyme. This enzyme propagates through a bidirectionally connected system, while producing one isomorphic copy of the whole system, using exclusively local actions on objects and their neighborhoods.

The figure on the right hand side shows a sequence of systems, each one resulting from its predecessor by the action of some specific fundamental enzyme. The whole chain of enzymes constitutes the *SplitFork* enzyme. The *SplitFork* propagates through the system, while copying the local structure it is attached to. The figure shows the explicit copying of the second object from the left and its local neighborhood. First, the *Apply* enzyme (aPPL) and the *CopyObjectInEnvironment* enzyme (!CPO) are applied. They initialize the fork list and copy the object containing the fork, respectively. Then, after applying the *PresentValenceIf* enzyme (!PRV), which presents the identity links between original object and copied object, the *ValenceMultLeftIf* (!VML) and the *AdjointMultRightIf* (!AMR) enzymes are applied. They detect links without adjoint, and declare composite arrows as new fundamental ones by motion with memory. The *ReportIf* enzyme (!RPT) searches for links with missing adjoint, while the *RenderFundamental* enzyme (!MFU) renders them into the system. The *PresentSource* enzyme (!PRS) declares the copied object as target of enzyme-actions, while the *MakeAdjoint* enzyme (!MAD) creates a new adjoint link for the copied object. Finally, the *RemoveAdjoint* (!RMA) and the *RemoveValence* (!RMV) enzymes remove the identity links between original and copied object. As seen, this procedure copies the local structure. Proceeding to the new forks in the system, the enzyme sequence finally copies the whole system [30].



Appendix B

CONSTRUCTING THE DIRAC SYSTEM

In [59] Kogut and Susskind proposed a discretized version of the Dirac equation on a lattice. The discretization is performed on a three-dimensional cubic space-lattice, with lattice spacing a. Denoting each lattice site by the vector $\vec{r}$, the Kogut-Susskind discretization for the two-component spinor $\psi(r)$ on the lattice, is given by

$$\imath\dot{\psi} = \frac{1}{\imath a}\sum_{\vec{n}}(\vec{\sigma}\cdot\vec{n})\ \psi(\vec{r}+\vec{n})\ +\ m_0(-1)^{r_x+r_y+r_z}\psi(\vec{r}) \quad , \tag{B.0.1}$$

where m_0 is the mass of the fermion and $\vec{\sigma}$ the Pauli matrices. The $\vec{n}$ denotes the link vector to the six next-neighbors of the current site. The factor $(-1)^{r_x+r_y+r_z}$ grades the lattice sites and is important for massive fermion fields (staggered fermions). However, here the consideration is restricted to massless fermions, thus the second term on the right hand side of (B.0.1) vanishes. Moreover, the time is thought to be discrete, too, the time step being τ, and equation (B.0.1) becomes

$$\psi(X,t+\tau) = \psi(X,t)\ +\ \sum_{\hat{k}}(\vec{\sigma}\cdot\hat{k})\ \psi(X+\hat{k},t) \quad , \tag{B.0.2}$$

where $\hat{k} = \vec{k}/a$ is the normalized link vector from X to $X+\vec{k}$. Introducing the parallel transporters

$$\eta^k(X) := \vec{\sigma}\cdot\hat{k}(X)\ , \tag{B.0.3}$$

fulfilling

$$\eta^{-k} = \eta^k \tag{B.0.4}$$

one gets the equation

$$\psi(X,t+\tau) = \psi(X,t)\ +\ \tau\sum_{k=\pm1,\pm2,\pm3}\eta^{-k}(X+\vec{k})\ \psi(X+\hat{k},t)\ . \tag{B.0.5}$$

The parallel transporters η fulfill the following algebra

$$\begin{aligned} \eta^i(X+\hat{j})\eta^j(X) &= -\eta^j(X+\hat{i})\eta^i \quad i \neq \pm j && \text{(B.0.6)} \\ \eta^{-i}(X+\hat{j})\eta^i(X) &= -1 && \text{(B.0.7)} \\ \eta^{2i} &= 1\ . && \text{(B.0.8)} \end{aligned}$$

By using the identity

$$\sigma^i\sigma^j = \delta^{ij} + \imath\epsilon^{ijk}\sigma^k$$

for the Pauli matrices, it is easy to see, that the algebra of the parallel transporters holds for a general three-dimensional regular lattice, too, supposed (B.0.4) is fulfilled. Therefore, equation (B.0.5) may be considered on a regular three-dimensional lattice, too.[1]

Equation (B.0.5) is now used to construct a ToCS-based formulation. For this purpose, an appropriate system is defined constructively, and the universal dynamics corresponding to equation (B.0.5) is investigated.

B.1 THE DIRAC SYSTEM

In order to define a ToCS-based formulation of the Dirac equation, the system $\mathcal{S}_D$ is constructed. For this, suppose a regular space lattice Γ with sites $V(\Gamma)$ and edges $E(\Gamma)$ is given. The parallel transporters are attached to the edges $E(\Gamma)$ by the formal products $e \otimes \eta$. Assigning to each site X its linear state space Ω_X, the objects of the category $\mathcal{K}_D$ are¡defined by adding the target space Ω_∞ of the Dirac field ψ to the set of state spaces $\{\Omega_X\} \cong V(\Gamma)$

$$\text{Ob}(\mathcal{K}_D) \ \cong\ V(\Gamma) \cup \{\Omega_\infty\}\ . \qquad \text{(B.1.1)}$$

The Dirac field ψ is implemented by links from Ω_∞ to sites Ω_X, the conjugated field ψ^* becomes links to Ω_∞

$$\begin{aligned} \psi(X) &: \Omega_\infty \to \Omega_X \\ \psi^*(X) &: \Omega_X \to \Omega_\infty\ . \end{aligned}$$

A corresponding set of arrows is generated as paths by the fundamental arrows

$$\text{Fund}(\mathcal{K}_D) := E(\Gamma) \cup \{\psi(X)\} \cup \{\psi^*(X)\}\ .$$

The set of paths generated from this set, is denoted by P. Its elements are called free paths (no constraints). To extract from P the appropriate set of arrows $\text{Ar}(\mathcal{K}_D)$ and thereby the category $\mathcal{K}_D$ and the system $\mathcal{S}_D$, constraints must be imposed on the set of paths. Heuristically, for an arbitrary arrow $f : \Omega_X \to \Omega_Y$ this is done by decomposing f into fundamental

[1] Actually, equation (B.0.5) represents the Dirac operator on the lattice. Defining it as the square root of the Laplacian, it is possible to choose the parallel transporters to be real or complex numbers, as shown in [107] for a cubic and for a triangular lattice.

arrows and selecting in a constructive way only its so called source-ordered irreducible components, which are paths from X to Y builded from links that generate f. This will be described in detail later. The constraint is then imposed by identifying f with the "sum" of its ordered irreducible subpaths.

Irreducible subpaths

Making the above statement more precisely, we introduce the notion $\tilde{f}$ for the value of the arrow f, that is, for the function

$$\tilde{f} : \Omega_X \to \Omega_Y \quad ,$$

that corresponds to f. Of course, given two arrows $f, g : \Omega_X \to \Omega_Y$ their sum $\tilde{f} + \tilde{g}$ is well defined for linear spaces Ω_Y.

Since every free path is generated by definition from elements of Fund$(\mathcal{K}_D)$, there exists a unique decomposition (modulo identities)

$$f \;=\; \prod_{i=1}^{n} b_i \quad . \tag{B.1.2}$$

Now, let $\phi : X \to Y$ be another path having same source and target as f and consisting of a subfamily $(b_j)_{j\in J}$ of the (b_i) with $J \subset \{1, ..., n\}$ being an ordered subset

$$\phi = \prod_{j\in J} b_j \quad .$$

Suppose further, that there exists loops $(\lambda_a)_{a=0,1,...,p}$ at the objects $\alpha(b_{k_a})$ respectively given for all k_a by

$$\lambda_a \;:=\; \prod_{j_a=k_a}^{k_a+l_a} b_{j_a} \quad ,$$

with $\{k_a, ..., k_a+l_a\} \cap J = \emptyset$. We have set $k_0 = l_0 = 0$ and $k_p = n$. The first and the last loop are defined consistently as $\lambda_0 := \iota_{\alpha(f)}$ and $\lambda_p := \iota_{\omega(f)}$, since this is the only ambiguity in (B.1.2). Then, we define:

> The path ϕ is called a **subpath** of f, if the families $(b_{j_a})_{j_a=k_a,...,k_a+l_a}$ fulfil the properties
>
> $$(i) \quad \Big[\mathrm{Ob}(\lambda_a) \cap \mathrm{Ob}(\phi)\Big] \setminus \{\alpha(\lambda_a)\} \neq \emptyset \quad \text{, for } a = 1, ..., p-1$$
>
> $$(ii) \quad \phi \;=\; \prod_{a=0}^{p-1} \prod_{j=k_a+l_a+1}^{k_{a+1}} b_j$$
>
> $$(iii) \quad f \;=\; \prod_{a=0}^{p-1} \left(\prod_{j=k_a+l_a+1}^{k_{a+1}} b_j \;\circ\; \lambda_a \right) \quad .$$

Operationally, the subpaths of f are given by extracting loops out of the product (B.1.2) which due to property (i) have at least two objects in common with the "foregoing" links. Note that, from properties (ii) and (iii) it follows, that f is a subpath of itself.
A subpath ϕ of f is called a **proper subpath**, if $\phi \neq f$.

In this sense, the subpaths of a given path f describe the sub-products in (B.1.2), that travel from the source of f to its target, respecting the order of the generating links. A path, that do not contain any proper subpath is called **irreducible**, otherwise it is called a **reducible** path. In analogy subpaths are called irreducible, if they are irreducible paths.

For the definition of the system $\mathcal{S}_D$ a specific set of irreducible subpaths called the source-ordered irreducible subpaths of f is considered. Let f be given by the product (B.1.2). We separate this product into the factors

$$f \;=\; \prod_{j=k_1+1}^{n} b_j \;\circ\; \left(\prod_{i=1}^{k_1} b_i\right) \quad , \tag{B.1.3}$$

the term in brackets being the first reducible subproduct starting with b_1 at the source $\alpha(f)$. This is a well-defined separation. Denote by ρ_s its irreducible subpaths and consider the products

$$\phi_s := \prod_{i=k_1+1}^{n} b_i \quad \circ\; \rho_s \tag{B.1.4}$$

that is, replace the first reducible part in (B.1.3) (i.e. the term in brackets) by its irreducible subpaths. Now take the products ϕ_s of (B.1.4) and proceed in the same manner by replacing their first reducible parts starting with ρ_s by their corresponding irreducible subpaths. Since the product (B.1.2) is finite this procedure terminates after a finite number of steps, producing a subset of the irreducible subpaths of f called the **source-ordered irreducible subpaths** $I_\alpha(f)$.

The set $I_\alpha(f)$ is by definition "non-symmetrical" in the sense, that it distinguishes the source object $\alpha(f)$. Proceeding analogously, this definition may be reversed, that is, one may start with b_n at the target, replacing the first reducible subproduct by its irreducible subpaths. This would yield a modified constraint when applied to arbitrary paths. Nevertheless both constraints produce the same result when applied to the case that is essential for the formulation of the Dirac equation, so it is possible to chose among both versions.
This asymmetry will be faced again later, where an appropriate composition law guaranteeing a homomorphic behavior of the constraint is defined.

Since $I_\alpha(f)$ is achieved by replacing the reducible parts of f by their ordered irreducible subpaths, some of the objects passed by f may not be reached by any irreducible subpath in $I_\alpha(f)$. As a consequence, given two composable but otherwise arbitrary paths f and g,

we get

$$I_\alpha(f \circ g) \neq I_\alpha(f) \circ I_\alpha(g) \quad , \tag{B.1.5}$$

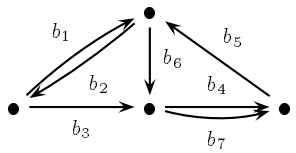
in general, where the composition on the right hand side is understood for every pair in $I_\alpha(f) \times I_\alpha(g)$. To see this, suppose equality holds in (B.1.5) for every path and consider its decomposition

$$I_\alpha(f) = I_\alpha\left(\prod b_i\right) = \prod I_\alpha(b_i) \quad .$$

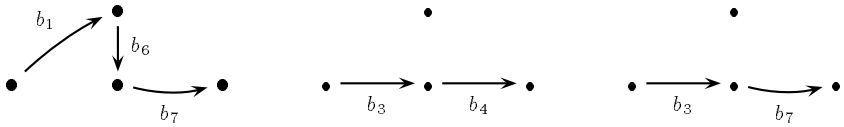
Since every fundamental arrow b_i is irreducible, this would imply that f is irreducible, which is not true in general. This kind of non-homomorphic behavior of $I_\alpha(f)$ gives rise to the construction bellow.

Examples

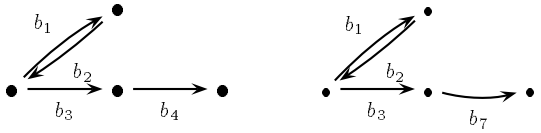
To illustrate the introduced notion of irreducible paths, consider the path f, defined as follows ($f := b_7 \circ \ldots \circ b_1$)



Its source-ordered irreducible subpaths $I_\alpha(f)$ are given by



Now separate the path f into the factors $h := b_3 \circ b_2 \circ b_1$ and $g := b_7 \circ b_6 \circ b_5 \circ b_4$. Since h is irreducible, the only reducible paths arise in g. The ordered irreducible subpaths of g are $I_\alpha(g) = \{b_4, b_7\}$. Composing them with h yields the set $I_\alpha(g) \circ I_\alpha(h)$ consisting of the elements



which implies

$$I_\alpha(f) \;\neq\; I_\alpha(g) \circ I_\alpha(h) \quad ,$$

as expected from the remark above.

Formulating the ToCS-based analogon of the Dirac equation, we wish to identify a path with the sum of its ordered irreducible subpaths. Demanding this property naively, results incompatibilities with the composition law, because of the non-homomorphic behavior of the sets $I_\alpha(f)$. Consider the paths of the previous example f, g and h with $f = g \circ h$. Identifying naively the corresponding functions $\tilde{f}$, $\tilde{g}$ and $\tilde{h}$ with the sum of their ordered irreducible subpaths respectively, the property (B.1.5) would imply

$$\sum_{\phi \in I_\alpha(f)} \tilde{\phi} \neq \sum_{\xi \in I_\alpha(g)} \tilde{\xi} \quad \circ \sum_{\chi \in I_\alpha(h)} \tilde{\chi} \quad , \tag{B.1.6}$$

where distributivity is assumed for the right hand side to be evaluated.[2] The reason for this discrepancy lies in the fact, that in building the set $I_\alpha(f)$ the irreducible subpaths of h are composed with the whole g, not only with its ordered irreducible subpaths as in $I_\alpha(g) \circ I_\alpha(h)$. To bypass this obstruction, we wish to turn the construction of the ordered irreducible subpaths into an "invertible" operation carrying information about the original path f. This would enable the reconstruction of the whole $I_\alpha(f)$ and is realized by considering the pairs $(f, \tilde{\phi})$ instead of $\tilde{\phi}$ for subpaths ϕ of f. To be able to respect the path building property of the composition of arrows $\circ$, a composition law $\odot$ must be defined for pairs $(f, \tilde{\phi})$ and $(g, \tilde{\chi})$ with composable f and g.

Extending the set of arrows

More precisely, let $P(f)$ be the set of all subpaths of f and denote by $P_+(f)$ the set $P(f)$ with its elements considered as additive functions, equipped with the ordinary addition of functions. We define the bundle

$$B \;:=\; \bigcup_{f \in P} \{f\} \times P_+(f) \tag{B.1.7}$$

where by P we denote the set of all paths generated by Fund$(\mathcal{K}_D)$. Note that $P_+(f)$ contains not only the subpaths of f but also sums of their corresponding functions. The canonical projection π of B is given by

$$\begin{aligned} \pi : \quad B \quad &\to P \\ (f, \tilde{\phi}) \quad &\mapsto f \quad . \end{aligned} \tag{B.1.8}$$

Since every path is a subpath of itself, the set P is canonically embedded into B via the section $\mathcal{B}$, satisfying $\pi \circ \mathcal{B} = \mathrm{id}_P$

$$\begin{aligned} \mathcal{B} : P \quad &\hookrightarrow \quad B \\ f \quad &\mapsto \quad (f, \tilde{f}) \quad . \end{aligned} \tag{B.1.9}$$

[2] Strictly, the right hand side of (B.1.6) is ill defined, since $\circ$ is not a composition law for functions $\tilde{\phi}$. Stating heuristically, the construction following in the text introduces an effective composition $\odot$ for "functions", bypassing the property (B.1.6).

For the constraint to be imposed we introduce the section

$$\begin{array}{rcl} \mathcal{C} \;:\; P & \to & B \\ f & \mapsto & \Big(f,\; \displaystyle\sum_{\phi \in I_\alpha(f)} \tilde{\phi} \;\Big) \end{array} \quad . \tag{B.1.10}$$

Restricting $\mathcal{C}$ onto its image $\mathcal{C}[P]$ this becomes an invertible map, due to $\pi \circ \mathcal{C} = \mathrm{id}_P$.

From the definition of subpaths follows that $I_\alpha(f) = \{f\}$, for all irreducible paths f. This implies the relation $\mathcal{B}(f) = \mathcal{C}(f)$ for irreducible paths. We extend this property to the whole set P by stating the constraint

$$\mathcal{B}(f) = \mathcal{C}(f) \tag{B.1.11}$$

for every path $f \in P$, that is, we identify the sections $\mathcal{C}[P]$ and $\mathcal{B}[P]$ of fig. B.1 respecting the projection π of the bundle.

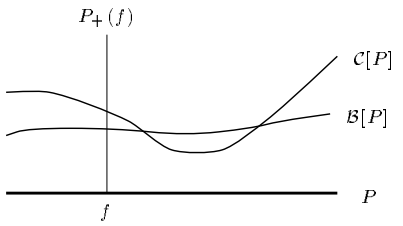


Figure B.1: The bundle B

Applying the constraint (B.1.11) to an arbitrary path f and restricting attention on the second component of $\mathcal{B}(f)$ and $\mathcal{C}(f)$ then yields the constraint for the linear functions

$$\tilde{f} \;=\; \sum_{\phi \in I_\alpha(f)} \tilde{\phi}$$

and thereby the desired identification of the corresponding paths.

The implementation of the section $\mathcal{C}$ as a homomorphism guarantees the compatibility of path building and applying the constraint. This is achieved by introducing an appropriate structure on the bundle B. Given elements $p = (f, \tilde{\phi})$ and $q = (g, \tilde{\chi})$ of the bundle with composable projections $\pi(p)$ and $\pi(q)$, we define their composition $p \odot q$ to be

$$p \odot q \;:=\; \Big(f \circ g \;,\; \sum_{\xi \in I_\alpha(f \circ \chi)} \tilde{\xi} \;\Big) \quad . \tag{B.1.12}$$

As mentioned earlier this composition law does not treat the points p and q symmetrically. It does not depend on the point in the fibre of p. This is due to the fact, that we have chosen

the *source-ordered* irreducible subpaths for the constraint to be defined. Taking instead the *target ordered* version would yield a composition law independent of the fibre of the second point q.

For the associativity of the composition (B.1.12), note first the inductive property

$$I_\alpha(f \circ g) \;=\; \bigcup_{\zeta \in I_\alpha(g)} I_\alpha(f \circ \zeta) \quad , \tag{B.1.13}$$

of the sets $I_\alpha(f)$. It follows directly from the definition. Equation (B.1.13) then implies

$$\begin{aligned}
\Big[(f,\tilde{\phi}) \odot (g,\tilde{\chi})\Big] \odot (h,\tilde{\xi}) &= \Big((f\circ g)\circ h \;,\; \sum_{\theta \in I_\alpha(f\circ g\circ \xi)} \tilde{\theta}\Big) \\
&= \Big(f\circ(g\circ h) \;,\; \sum_{\zeta\in I_\alpha(g\circ\xi)} \sum_{\theta\in I_\alpha(f\circ\zeta)} \tilde{\theta}\Big) \\
&= (f,\tilde{\phi}) \odot \Big[(g,\tilde{\chi}) \odot (h,\tilde{\xi})\Big] \quad ,
\end{aligned}$$

that is, the associativity of the composition law $\odot$ for $p,q,r \in B$

$$(p \odot q) \odot r = p \odot (q \odot r) \quad .$$

In a similar manner equation (B.1.13) yields

$$\mathcal{C}(f \circ g) \;=\; \mathcal{C}(f) \odot \mathcal{C}(g) \quad , \tag{B.1.14}$$

for the composition of elements $\mathcal{C}(f)$ and $\mathcal{C}(g)$ in B, turning the constraint map $\mathcal{C}$ into a homomorphism with respect to the composition law $\odot$.
Stated differently, equation (B.1.14) shows, that the composition structure on B induces consistently a composition structure on $\mathcal{C}[P]$. The associativity of the constraint follows trivially from the associativity of the composition of paths.

It is easy to check the consistency relation

$$\mathcal{B}(f) \odot \mathcal{B}(g) = \mathcal{C}(f) \odot \mathcal{C}(g) \quad . \tag{B.1.15}$$

The properties (B.1.14) and (B.1.15) ensure the compatibility of the composition law and the constraint (B.1.11). They enable the evaluation of the constraint on every part of a given path.

The sets $\mathcal{B}[P]$ and $\mathcal{C}[P]$ are used to define the arrows of the desired system $\mathcal{S}_D$. Consider first their union $\mathcal{B}[P] \cup \mathcal{C}[P]$. This set closes under the composition $\odot$, as is easily seen from the following relations:

$$\begin{aligned}
\mathcal{C}(f) \odot \mathcal{C}(g) &= \mathcal{C}(f\circ g) \\
\mathcal{B}(f) \odot \mathcal{B}(g) &= \mathcal{C}(f\circ g) \\
\mathcal{B}(f) \odot \mathcal{C}(g) &= \mathcal{C}(f\circ g) \\
\mathcal{C}(f) \odot \mathcal{B}(g) &= \mathcal{C}(f\circ g) \quad .
\end{aligned}$$

Moreover, the set $\mathcal{C}[P]$ is generated from $\mathcal{C}[\text{Fund}(\mathcal{K}_D)]$, since $\mathcal{C}$ is a homomorphism. Including formal identities at every object, the implementation of the constraint (B.1.11) remains the only step towards the system $\mathcal{S}_D$. For fundamental arrows, and more general, for every irreducible path, the constraint is fulfilled by definition. The only condition is imposed for those arrows $\mathcal{B}(f)$, which originate from irreducible, free paths f as in (B.1.2). The constraint requires, these arrows to be generated by the $\mathcal{C}(b_i)$

$$\mathcal{B}\Big(\prod_i b_i\Big) = \odot_i \; \mathcal{C}(b_i) \;\; , \tag{B.1.16}$$

that is, the set of arrows $\mathcal{C}[\text{Fund}(\mathcal{K}_D)]$ should generate the whole set of arrows. This construction is summarized in the following result:

Result (Dirac-System):
The Dirac-System $\mathcal{S}_D$ consists of objects Ob$(\mathcal{K}_D)$ defined in (B.1.1). The arrows of the system are given by the union $\mathcal{C}[P] \cup \mathcal{B}[P]$ generated from the fundamental arrows

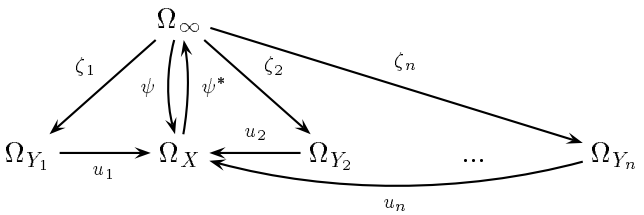
$$\mathcal{C}[\textit{Fund}(\mathcal{K}_D)] \tag{B.1.17}$$

and the formal identities at the objects by the composition law $\odot$ defined in (B.1.12), due to the constraint (B.1.16).

With this we can now proceed to the formulation of the Dirac equation.

B.2 THE DIRAC EQUATION

In order to proceed to a ToCS-based formulation of equation (B.0.5), it is necessary to find paths in the Dirac system $\mathcal{S}_D$, that correspond to this equation under the constraint (B.1.16). The appropriate paths are constructed in the following scenario



where it is supposed, that all objects are different from each other. The arrows are contained in the original set Fund$(\mathcal{K}_D)$.

In the diagram above, the loops $l_k : \Omega_\infty \to \Omega_\infty$ are defined as follows

$$l_k := \psi^* \circ u_k \circ \zeta_k \;\; ,$$

where $\zeta_k : \Omega_\infty \to \Omega_{Y_k}$ and $u_k : \Omega_{Y_k} \to \Omega_X$ are paths that do not intersect each other outside their common object Ω_{Y_k}.
From the loops l_k the path d is defined by

$$d := \psi \circ \prod_{k=1}^{n} l_k \quad .$$

The path d is embedded into the Dirac system by the section $\mathcal{C}$. To calculate $\mathcal{C}(d)$, note, that the ordered irreducible subpaths of d are given by

$$I(d) = \{\psi\} \cup \{u_k \circ \zeta_k \ : \ k = 1, ..., n\} \quad ,$$

which yields immediately

$$\mathcal{C}(d) = \left(d \, , \tilde{\psi} + \sum_{k=1}^{n} \widetilde{u_k \circ \zeta_k}\right) \quad .$$

Applying the constraint (B.1.16) to the path d, leads to the equation

$$\left(\widetilde{\psi \circ \prod_{k=1}^{n} l_k}\right) = \tilde{\psi} + \sum_{k=1}^{n} \widetilde{u_k \circ \zeta_k} \quad . \tag{B.2.1}$$

Substituting in equation (B.2.1) for u_k the link from the next neighbor of X in $\hat{k}$-direction to X and for ζ_k the Dirac link $\psi(X + \hat{k})$

$$\begin{aligned} u_k(X) &:= \eta^{-k}(X + \hat{k}) \\ \zeta_k(X) &:= \psi(X + \hat{k}) \end{aligned}$$

the loops l_k are given by all fundamental triangles containing the link $\psi^*(X)$. With this, equation (B.2.1) takes the form

$$\psi(X) \circ \prod_{k=1}^{n} l_k(X) = \psi(X) + \tau \sum_{k=1}^{n} \eta^{-k}(X + \hat{k}) \, \psi(X + \hat{k}) \quad , \tag{B.2.2}$$

where we have neglected the $\tilde{}$-signs. The right hand side of (B.2.2) is identical to the right hand side of (B.0.5). From this, the dynamics of the field ψ may be stated in ToCS-based notion as follows:

Result (Dirac-Dynamics):
In the Dirac system $\mathcal{S}_D$, the universal dynamics law

$$\psi(X, t + \tau) = \psi(X) \circ \prod_{k=\pm1,\pm2,\pm3} l_k(X) \quad , \tag{B.2.3}$$

corresponds via the constraint (B.1.16) to the Kogut-Susskind discretization of the Dirac equation with discretized time $t \in \tau\mathbb{Z}$.

Being a universal dynamics law, equation (B.2.3) translates easily into graphical notation. Abstracting the scenario from the beginning of the section, the universal dynamics is shown in fig. B.2.

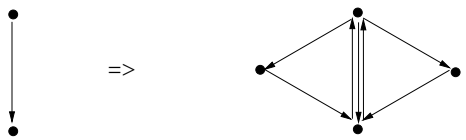


Figure B.2: The universal dynamics of the Dirac equation. The triangles are traveled first, this is possible in arbitrary order.

However, although the universal dynamics (B.2.3) and the Dirac system implement the Kogut-Susskind discretization, the constraint (B.1.16) may further restrict the arrows in $\mathcal{S}_D$. In this sense, the consideration here realizes the minimal condition to include a ToCS-based version of the Dirac equation in the dynamics of the system $\mathcal{S}_D$.

Appendix C

THE UML NOTATION

C.1 VIEWS AND DIAGRAMS

In chapter 4, Kruchten's 4+1 view approach was mentioned as architectural basis for the UML. The 4+1 views in this approach specify both, the structure of the metamodel that defines the view-constituents in corresponding packages, and the way a UML-model may be described by using of diagrams that represent the views graphically, while corresponding to the metamodel definitions.

The graphical representation of UML-models are notated by specific diagrams.

> A **diagram** describing a view is a (connected) graph whose edges represent relations contained in the view, and whose vertices represent model elements in the view.

There are some standard diagrams predefined in the UML which are used in the model views for their graphical representation:

- **Logical View**
  The structural aspects of a logical view are shown by *class diagrams*, whereas for the behavioral aspects *interaction diagrams* and *statecharts* are used.
- **Process View**
  For a process view *class diagrams* are used for structural and *interaction diagrams* for behavioral aspects.
- **Implementation View**
  The implementation view concerning primarily with the software module architecture is represented by *component diagrams*.

- **Deployment View**
  The deployment view concentrates on the hardware issues of a information system, as given usually by a client/server architecture and is notated by corresponding *deployment diagrams*.

- **Use Case View**
  The use cases of a model are shown in *use case diagrams* and in form of *activity diagrams* for the behavioral aspects of the use cases.

C.2 GRAPHICAL SYMBOLS

Here the most important symbols for a graphical representation of model elements are presented. See [85, 86, 89, 87, 76, 75, 108] for details.

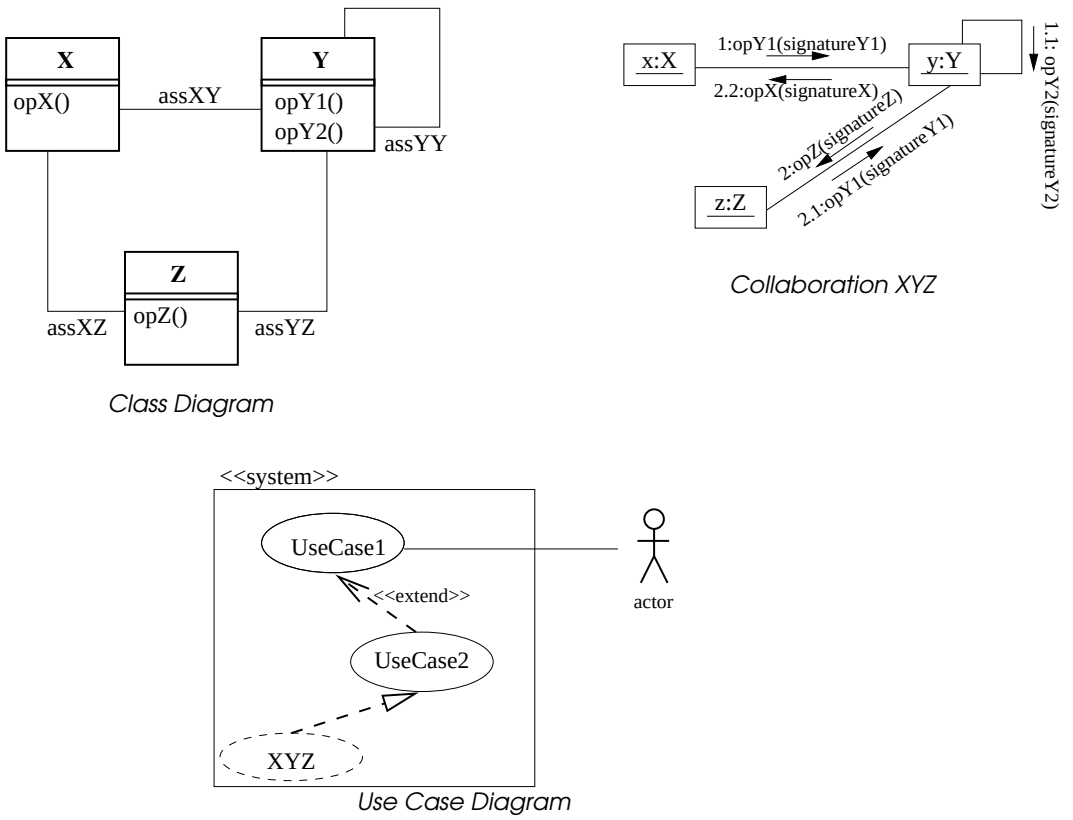


Figure C.1: Class, Collaboration, and Use Case diagrams in the UML.

Appendix D

TWO PROOFS

This section provides the proofs of the Main Representation Theorem and of the Communication Network Construction of chapter 1 as presented in [28].

> **Main Representation Theorem:**
> *Every small category $\mathcal{K}$ admits a faithful representation on a subcategory $\mathcal{C}$ of the category of sets* **Set**.

To prove this, for an arbitrary small category $\mathcal{K}$ its so called **communication network** is constructed out of the given data. The main representation theorem follows from this construction as a corrolary.

> **Communication Network Construction:**
> *Given a category $\mathcal{K}$ there exists for every $X \in Ob(\mathcal{K})$ sets A_X (input space) and Ω_X (output space) and a correspondence $\tilde{}$ which assigns to each X an invertible map*
>
> $$\tilde{X} : A_X \to \Omega_X \tag{D.0.1}$$
>
> *and to each $f \in Ar(X, Y)$ a map*
>
> $$\tilde{f} : \Omega_X \to A_Y \tag{D.0.2}$$
>
> *with the properties*
>
> $$\tilde{X} \circ \tilde{\iota}_X = id_{\Omega_X} \quad and \quad \tilde{\iota}_X \circ \tilde{X} = id_{A_X} \tag{D.0.3}$$
>
> $$\widetilde{g \circ f} = \tilde{g} \circ \tilde{Y} \circ \tilde{f} : \Omega_X \to A_Z \, . \tag{D.0.4}$$

Proof (Communication Network Theorem):
We first define the sets of incoming and outgoing arrows at an object X respectively as

$$\mathrm{In}_X \quad := \quad \mathrm{Ar}(*, X) := \bigcup_{Z \in \mathrm{Ob}(\mathcal{K})} \mathrm{Ar}(Z, X)$$

$$\mathrm{Out}_X \;:=\; \mathrm{Ar}(X,*) := \bigcup_{Z\in \mathrm{Ob}(\mathcal{K})} \mathrm{Ar}(X,Z)$$

We introduce an equivalence relation on the sets In_X. Let $f, g \in \mathrm{In}_X$, we set

$$f \sim g \;:\Leftrightarrow h\circ f = h\circ g \quad \forall h \in \mathrm{Out}_X$$

and define the input space as the quotient

$$A_X := \mathrm{In}_X/\sim \;\; .$$

Consider the set of maps

$$\bar{\Omega}_X := \{\zeta : \mathrm{Out}_X \to \mathrm{Ar}(\mathcal{K}) \;: \zeta(f) \in \mathrm{In}_{\omega(f)}\}$$

where $\omega(f)$ again denotes the target of $f \in \mathrm{Ar}(X,Y)$, that is $\omega(f) = Y$. We define a map

$$\begin{aligned} \tilde{X} : \mathrm{In}_X &\to \bar{\Omega}_X \\ f &\mapsto \tilde{X}f \end{aligned}$$

acting on Out_X via

$$\tilde{X}f(g) := g\circ f \;\; .$$

The output space Ω_X is then defined to be the image of $\tilde{X}$

$$\Omega_X := \mathrm{im}\tilde{X} \subset \bar{\Omega}_X$$

and $\tilde{X}$ becomes an invertible map on the in- and output space

$$\tilde{X} : A_X \to \Omega_X \;\; ,$$

establishing (D.0.1).
To construct the further data we define

$$\tilde{\iota}_X := \tilde{X}^{-1}$$

so that the relations (D.0.3) are fulfilled.
The correspondence $f \mapsto \tilde{f}$ for $f \in \mathrm{Ar}(X,Y)$ is defined with the help of the $\tilde{\iota}_X$ by

$$\tilde{f} := \hat{f}\circ\tilde{\iota}_X \;\; : \Omega_X \to A_Y$$

where $\hat{f}$ is the map

$$\begin{aligned} \hat{f} : A_X &\to A_Y \\ [g] &\mapsto [f\circ g] \;\; , \end{aligned}$$

so that (D.0.2) is fulfilled.
Property (D.0.4) is easy to calculate with the help of (D.0.3).

□

Now it is possible to proceed to the proof of the main representation theorem.

Proof (Main Representation Theorem):
Given the communication network data of the category $\mathcal{K}$ we define the representation functor ρ as follows:

$$\begin{aligned} \rho(X) &:= \Omega_X \in \mathrm{Ob}(\mathbf{Set}) \\ \rho(f) &:= \tilde{Y} \circ \tilde{f} \end{aligned}$$

for every $X \in \mathrm{Ob}(\mathcal{K})$ and $f \in \mathrm{Ar}(X, Y)$. The functor ρ is covariant, with $g \in \mathrm{Ar}(Y, Z)$ we have

$$\begin{aligned} \rho(g \circ f) &= \tilde{Z} \circ \widetilde{(g \circ f)} \\ &= \tilde{Z} \circ \tilde{g} \circ \tilde{Y} \circ \tilde{f} \\ &= \rho(g) \circ \rho(f) \end{aligned}$$

and therefore a representation onto the image $\mathcal{C}$ of ρ in **Set**.

□

Bibliography

[1] N. Luhmann, *Soziologische Aufklärung 2.* Westdeutscher Verlag, Opladen (1975).

[2] M. M. Waldrop, *Complexity: The Emerging Science at the Edge of Order and Chaos.* Simon and Shuster, New York (1992).

[3] T. Davenport, Putting the Enterprise into the Enterprise System, *Harvard Business Review* 4/1998 (1998).

[4] H. Haken, *Synergetics*. Springer, Heidelberg (1978).

[5] P. Bak, C. Tang and K. Wiesenfeld, Self-Organized Criticality, *Phys. Rev. Lett.* 59:381 (1989).

[6] P. Bak, C. Tang and K. Wiesenfeld, Self-Organized Criticality, *Phys. Rev. A* 38:364 (1988).

[7] G. Held et al., Experimental Study of Critical-Mass Fluctuations in an Evolving Sandpile, *Phys. Rev. Lett.* 65:1120 (1990).

[8] H. Z. Wojciech, Complexity, Entropy and the Physics of Information, in: *Santa Fe Institute Studies in the Sciences of Complexity*, Redwood City (1990). Addison-Wesley.

[9] D. Pines, Emerging Synthesis in Science, in: *Santa Fe Institute Studies in the Sciences of Complexity*, Addison-Wesley (1986). Proceedings vol. 1.

[10] I. N. Vorontsov, Description and Investigation of Biochemical Mechanisms of Living Cells, in: *Multiscale Phenomena, Modelling and Computation*, Eilat (1997). Research Workshop of the Israeli Science Foundation.

[11] M. Meier-Schellersheim and G. Mack, SIMMUNE: a tool for simulating and analyzing immune system behavior, preprint DESY-99-034, University of Hamburg (1999), submitted to J. Theor. Biol.

[12] A. S. Perelson and S. A. Kauffman, Molecular Evolution on Rugged Landscapes: Proteins, RNA, and the Immune System, in: *Santa Fe Institute Studies in the Sciences of Complexity*, Addison-Wesley (1990). Proceedings vol. 9.

[13] S. A. Kauffman and R. G. Smith, Adaptive Automata Based on Darwinian Selection, *Physica D* 22:68 (1986).

[14] G. Evans, *The Varieties of Reference*. Oxford, New York (1982).

[15] P. W. Anderson, K. J. Arrow and D. Pines., The Economy as an Evolving Complex System, in: *Santa Fe Institute Studies in the Sciences of Complexity*, Redwood City (1988). Addison-Wesley.

[16] W. B. Arthur, J. M. Ermoliew and J. M. Kaniowski, Path-Dependent Processes and the Emergence of Macro-Structure, *European Journal of Operational Research* 30:294 (1987).

[17] Y. C. Zhang, Modeling Market Mechanism with Evolutionary Games, preprint cond-mat/9803308, Fribourg (1998).

[18] H. Ulrich, Der systemorientierte Ansatz in der Betriebswirtschaftslehre, in: *Wissenschaftsprogramm und Ausbildungsziele der Betriebswirtschaftslehre*, Berlin (1971).

[19] P. E. Senge, Infrastructure for a Learning Organization, *Journal of Innovative Management* 2, No.2:4 (1996).

[20] R. S. Kaplan and D. P. Norton, Using the Balanced Scorecard as a Strategic Management System, *Harvard Business Review* 74:75 (1996).

[21] B. Carlsson and R. Stankiewicz, On the Nature, Function and Composition of Technological Systems, *Journal of Evolutionary Economics* 1 (2):93 (1991).

[22] T. Parson, *The social system.* Routledge & Kegan Paul, London (1964).

[23] N. Luhmann, *Soziale Systeme. Grundriss einer allgemeinen Theorie.* Suhrkamp, Frankfurt (1984).

[24] H. Willke, *Systemtheorie I: Grundlagen.* UTB Lucius & Lucius, Stuttgart (1996).

[25] T. Malsch, Socionic, talk at the Informatik Kolloqium, University of Hamburg (09.11.1998).

[26] N. Wiener, *Kybernetik.* Rowohlt, Hamburg (1969).

[27] C. G. Langton, Artificial Life, in: *Santa Fe Institute Studies in the Sciences of Complexity*, Redwood City (1989). Addison-Wesley.

[28] G. Mack, Universal Dynamics of Complex Adaptive Systems: Gauge Theory of Things Alive (DESY-94-075) (1994).

[29] G. Mack, Gauge theory of things alive, *Nucl. Phys.* B42:923 (1995).

[30] G. Mack, Gauge Theory of Things Alive: Universal Dynamics as a Tool in Parallel Computing, *Progr. Theoret. Phys. Suppl.* 122:201 (1996).

[31] G. Mack, Pushing Einstein's Principles to the Extreme, in: *Quantum Fields and Quantum Space Time*, edited by G. t'Hooft et al., New York (1997). Plenum Press. NATO-ASI series B: Physics vol. 364 (preprint gr-qc/9704034).

[32] M. Speh, Object-Oriented Literate Design of a Multigrid Toolkit for Lattice Gauge Theory Simulations, Phd-thesis, University of Hamburg (1994).

[33] J. Würthner, ToCS/SyCL, Phd-thesis, University of Hamburg (1999), in preparation.

[34] G. Mack, To Gauge Theory from a Minimum of a priori Structure. to appear in Proc. Steklov Math. Institute, special issue f. L. Faddeev (1999).

[35] K. Ilinski, Gauge Theory of Arbitrage, working paper IPHYS-1-97, IPhys Group (1997).

[36] K. Ilinski, Physics of Finance, preprint hep-th/9710148 (1997).

[37] I. N. Vorontsov, Some Main Principles of Computer Modelling and Simulation System Development, in: *Multiscale Phenomena, Modelling and Computation*, Eilat (1997). Research Workshop of the Israeli Science Foundation.

[38] G. Mack, Glossary of Complex Systems, white paper, Universität Hamburg (1998).

[39] S. Lang, *Algebra.* Addison-Wesley, Massachusetts (1993).

[40] S. MacLane, *Categories for the Working Mathematician.* Springer, New York (1971).

[41] S. MacLane and I. Moerdijk, *Sheaves in Geometry and Logic.* Springer, New York (1992).

[42] M. Creutz, *Quarks, Gluons and Lattices*. Cambridge University Press, Cambridge (1983).

[43] L. Faddeev and A. Slavnov, *Gauge Fields. Introduction to Quantum Theory*. Benjamin/Cummings, Reading (1980).

[44] M. Henneaux and C. Teitelboim, *Quantization of Gauge Systems*. Princeton University Press, Princeton (1992).

[45] T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering* 2,4:308 (1976).

[46] C. J. Date, *An Introduction to Database Systems*. Addison-Wesley, Reading (1979).

[47] A. Finkelstein et al., Viewpoints: A framework for integrating multiple perspectives in system development, *Int. Jour. of Softw. Eng. and Knowl. Eng.* 2 (1):31 (1992).

[48] G. Engels, R. Heckel, G. Taentzer and H. Ehrig, A View-Oriented Approach to System Modelling based on Graph Transformations, technical report (tfs.cs.tu-berlin.de/publikationen), TU-Berlin, Berlin (1997).

[49] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformations. Volume 1: Foundation*. World Scientific, Singapore (1997).

[50] C. A. Petri, Kommunikation mit Automaten, Phd-thesis, Bonn, Schriften des Institutes für Instrumentelle Mathematik (1962).

[51] J. L. Peterson, *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ (1981).

[52] J. Padberg, H. Ehrig and L. Ribeiro, Algebraic high-level net transformation systems., *Mathematical Structures in Computer Science* 5:217 (1995).

[53] C. Ermel and A. Martini, A Taste of Categorical Petri Nets, technical report (tfs.cs.tu-berlin.de/publikationen) Nr. 96-9, TU-Berlin, Berlin (1996).

[54] J. Meseguer and U. Montanari, Petri nets are monoids, *Information and Computation* 88(2):105 (1990).

[55] L. D. Landau and E. M. Lifschitz, *Lehrbuch der Theoretischen Physik, Band II.* Akademie Verlag, Berlin (1992).

[56] J. D. Jackson, *Classical Electrodynamics.* Wiley & Sons, New York (1975).

[57] T. Weiland, Die Diskretisierung der Maxwell-Gleichungen, *Phys. Bl.* 42 Nr.7:191 (1986).

[58] P. Imhof, Zeitentwicklung der Störung von Lösungen eines Systems mittels Zufallspfad-Entwicklung, Diploma thesis, University of Hamburg (1997).

[59] J. Kogut and L. Susskind, Hamiltonian formulation of Wilson's lattice gauge theories, *Phys.Rev.* D 11:395 (1975).

[60] NeXT, Objective-C Manual, OPENSTEP documentation package, NeXT Software, Inc., (1996).

[61] S. Philippi, System modelling using Object-Oriented Pr/T-Nets, technical report 25/97, Universität Koblenz, Landau (1997).

[62] D. Harel and E. Gery, Executable Object Modeling with Statecharts., *IEEE Computer* 30/7:31 (1997).

[63] S. Sanoff, Why Modeling?, IntelliCorp White Paper, www.intellicorp.com, IntelliCorp, Inc., (1997).

[64] L. Belady and M. M. Lehman, *Program Evolution. Process of Software Change.* London: Academic (1985).

[65] S. A. Kauffman, *At home in the universe.* Oxford University Press, Oxford (1995).

[66] I. Jacobson, M. Christerson, P. Jonsson and G. Övergaard, *Object-Oriented Software Engineering. A Use Case Driven Approach*. Addison-Wesley Longman (1992).

[67] G. Booch, *Object-Oriented Analysis and Design with Applications.* Addison-Wesley Longman (1994).

[68] J. Rumbaugh, *Object-Oriented Modeling and Design.* Prentice Hall: Englewood Cliffs, NJ (1991).

[69] R. Wirfs-Brock, B. Wilkerson and L. Wiener, *Designing Object-Oriented Software.* Prentice Hall PTR, New Jersey (1990).

[70] J. Martin and J. J. Odell, *Object-Oriented Methods. A Foundation. A UML Edition.* Prentice Hall PTR, New Jersey (1997).

[71] W. R. LaLonde and J. R. Pugh, Subclassing $\neq$ Subtyping $\neq$ IsA, *Journal of Object-Oriented Programming* 01/1991:57 (1991).

[72] A. T. Borgida et al., Generalization/Specialization as a Basis for Software Specification, in: *On Conceptual Modelling*, edited by M. L. Brodie et al., New York (1984). Springer.

[73] K. G. Wilson, Ab initio quantum chemistry: a source of ideas for lattice gauge theorists, *Nucl. Phys. B Proc. Suppl.* 17 (1990).

[74] U. Rüde, *Mathematical and Computational Techniques for Multilevel Adaptive Methods.* Frontiers in Applied Mathematics 13 SIAM, Philadelphia (1993).

[75] P.-A. Muller, *Instant UML.* Wrox Press Ltd., Birmingham (1997).

[76] R. Burkhardt, *UML - Unified Modeling Language.* Addison-Wesley Longman, Bonn (1997).

[77] D. Harel, Statecharts: a visual formalism for complex systems, *Science of Computer Programming* 8:231 (1987).

[78] F. H. D. Coleman and S. Bear, Introducing Statecharts, or How to Use Statecharts in Object-Oriented Design., *IEEE Trans. Soft. Eng.* 1:9 (1992).

[79] S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis.* Englewood Cliffs, Prentice Hall, NJ (1988).

[80] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Software Development Process.* Addison-Wesley, to appear (1999).

[81] G. Booch, I. Jacobson and J. Rumbaugh, UML Semantics, version 1.1, documentation set http://www.rational.com/uml and OMG-ad/97-08-04, Rational Software (1 September 1997).

[82] P. Kruchten, The 4+1 View Model of Architecture, *IEEE Software* Sep. 95 (1995).

[83] B. Russel and A. N. Whitehead, *Principia Mathematica* . Cambridge University Press, Cambridge (1910).

[84] J. R. Abrial, Data Semantics, in: *IFIP Working Conference on Data Base Management, Cargese*, edited by J. Klimbie et al., Amsterdam (1974). North Holland.

[85] G. Booch, I. Jacobson and J. Rumbaugh, UML Notation Guide, version 1.1, documentation set http://www.rational.com/uml and OMG-ad/97-08-05, Rational Software (1 September 1997).

[86] M. Fowler and S. Kendall, *UML Distilled, Applying the Standard Object Modeling Language*. Addison-Wesley, Reading (1997).

[87] B. Oestereich, *Objektorientierte Softwareentwicklung mit der Unified Modeling Language.* Oldenbourg, Wien (1997).

[88] G. Booch, I. Jacobson and J. Rumbaugh, UML Proposal to the Object Management Group, version 1.1, documentation set http://www.rational.com/uml and OMG-ad/97-08-02, Rational Software (1 September 1997).

[89] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide.* Addison-Wesley (1998).

[90] B. Oestereich, Objektorientierte Geschäftsprozessmodellierung mit der UML, *OBJEKTspektrum* 2/1998 (1998).

[91] T. Tomkos, OO-BPM & UML, talk http://lienhard.desy.de/mackag/cs/dipldiss/OOGPMUML.ps.gz, University of Hamburg (Feb/23/1998).

[92] C. Atkinson, Meta-modeling for distributed object environments, in: *First International Enterprise Distributed Object Computing Workshop, EDOC'97*. IEEE Computer Society Press (1997).

[93] R. Geisler, M. Klar and C. Pons, Dimensions and Dichotomy in Metamodeling, technical report (tfs.cs.tu-berlin.de/publikationen) Nr. 98-5, TU-Berlin, Berlin (1998).

[94] OMG, Meta Object Facility, joint revised submission, OMG Document ad/97-08-15, Object Management Group, Inc., (1 September 1997).

[95] G. Booch, I. Jacobson and J. Rumbaugh, Object Constraint Language Specification, documentation set http://www.rational.com/uml and OMG-ad/97-08-08, Rational Software (1 September 1997).

[96] R. Breu et al., Towards a Formalization of the Unified Modeling Language, technical report TUM-I9726, TU München, Munich (1997).

[97] G. Övergaard, A formal approach to relationships in the unified modeling language, in: *PSMT- Workshop on Precise Semantics for Software Modeling Techniques in conjunction with ICSE 98*, Kyoto (1998).

[98] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (1995).

[99] A. M. Davis, *Software Requirements: Objects, Functions and States*. Prentice-Hall, Englewood Cliffs, NJ (1993).

[100] M. L. Minsky, *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, NJ (1967).

[101] H. Ehrig and F. Orejas, Integration and Classification of Data Type and Process Specification Techniques, *Bulletin EATCS: Formal Specification Column* 65 (1998).

[102] H. Ehrig and F. Orejas, Dynamic Abstract Data Types: An Informal Proposal, *Bulletin EATCS* 53:162 (1994).

[103] F. Parisi-Presicce and A. Pierantonio, An Algebraic Approach to Inheritance and Subtyping, in: *Proceedings ESEC 1991*. Springer LNCS 550 (1991).

[104] M. Grosse-Rhode, *Algebra Transformation Systems and their Composition*. Springer LNCS 1382, Heidelberg (1998).

[105] J. Martin and J. J. Odell, *Object-Oriented Methods. Pragmatical Considerations*. Prentice Hall, Englewood Cliffs, New Jersey (1996).

[106] J. Würthner, SyCL Documentation, white paper, University of Hamburg (1999), unpublished.

[107] B. Holm, Anwendungen systemtheoretischer Verallgemeinerungen von Mehrskalenalgorithmen, Diploma thesis, University of Hamburg (1996).

[108] T. Tomkos, Formal Modeling, talk http://lienhard.desy.de/mackag/cs/, University of Hamburg (Jun/18/1998).

Acknowledgements

First, I wish to thank my thesis advisor Prof.Dr. G.Mack for much more than constant support and nearly prophetic choice of the thesis subject. His thoughts, ideas and advice, contributed essentially to the development of the basic ideas. Further, his open-mindness has enabled many vague ideas to develop into feasible tasks.

I would also like to thank the whole 'Mack-group', namely York Xylander, Jan Würthner, Martin Meier-Schellersheim, Ute Kerres, Peter Imhof and Michael Bartels for many fruitful discussions and insights in many problems, a single brain is impossible to be firm with. In particular, I gratefully acknowledge Dr. X. for bringing into the discussion UML, which turned out to be essential for the thesis. Thanks to Martin Meier-Schellersheim for becoming a little more familiar with both, the point of view of an immunologist and of an OO-pragmatist. Jan Würthner and Ute Kerres helped much in developing the section on electrodynamics. In particular, I would like to thank Jan for his exciting, friendly and really close collaboration, including many ups and downs. He contributed much to my understanding of the connections between theoretical and implementation issues both, in the UML, and in the ToCS-based approach.

A warm and special thank you to Astrid Lindberg for her love, calm and honesty. Thank you for being all the things you are.

Finally, I gratefully acknowledge the Deutsche Forschungsgemeinschaft (DFG) for the financial support.