EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Systems engineering : a formal approach. Part III. Modeling methods

Citation for published version (APA):
Hee, van, K. M. (1993). *Systems engineering : a formal approach. Part III. Modeling methods*. (Computing science notes; Vol. 9311), (Systems engineering : a formal approach; Vol. 3). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1993

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Eindhoven University of Technology
Department of Mathematics and Computing Science

Systems Engineering: a Formal Approach
Part III : Modeling Methods

by
K.M. van Hee

93/11

Computing Science Note 93/11
Eindhoven, April 1993

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Information Systems Engineering: a Formal Approach

by

K.M. Van Hee

March 30, 1993

This report is part of a preliminary version of a book that will be published.

Contents

I System concepts 9

1 Introduction 11

2 Application domains 15

3 Transition systems 23

4 Objects 33

5 Actors 47

6 Specification language 63
- 6.1 Values, types and functions 63
- 6.2 Value and function construction 68
- 6.3 Predicates 71
- 6.4 Schemas and scripts 72

II Frameworks 81

7 Introduction 83

8 Transition systems framework 85

9 Object framework 93

10 Actor framework 103

III Modeling Methods 129

11 Introduction 131

12 Actor modeling 135
- 12.1 Making an actor model after reality 135
- 12.2 Characteristic modeling problems 146
- 12.3 Structured networks 162
- 12.4 Net transformations 167

13 Object Modeling 173
13.1 Making an object model after reality 175
13.2 Characteristic modeling problems 188
13.3 Transformations to other object frameworks 199

14 Object oriented Modeling 217

IV Analysis Methods 231

15 Introduction 233

16 Invariants 235
16.1 Place invariants 238
16.2 Computational aspects 252
16.3 Transition invariants 260

17 Occurrence graph 263

18 Time analysis 271

19 Simulation 283

V Specification Language 295

20 Introduction 297

21 Semantic concepts 301
21.1 Values and types 301
21.2 Functions 309

22 Constructive part of the language 313

23 Declarative part of the language 329
23.1 Predicates and function declarations 329
23.2 Schemas and scripts 333

24 Methods for function construction 339
24.1 Correctness of recursive constructions 339
24.2 Derivation of recursive constructions 344

25 Specification methods 351
25.1 Value types for complex classes 351
25.2 Specification of processors 357

A Mathematical notions 365

B Syntax summary 371

C Toolkit 375

Part III

Modeling Methods

Chapter 11

Introduction

In this book a *method* is a set of *guidelines* and *techniques* that can be used for the following tasks:

method

- Construction of a model for a system. We distinguish two cases:
  - Making a model after *reality*, i.e. the systems engineer has an informal description of the system, or he formed an idea of the system in mind base on observations of the system, and out of this he makes a model in terms of the formalisms given in the preceding part.
  - *Transforming* a given model into another one, in which case the second model is formulated in another formalism or in the same formalism but with more structure, (i.e. has properties the first model does not have) or formulated differently.
- Analysis of a model. Again we distinguish two cases:
  - *Verifying* properties of a model by means of a formal proof.
  - *Validating* a model by means of *simulation* of the behavior of the model, i.e. testing hypotheses or calculating characteristic values of a model based on the simulation experiments.

model making

Making a model from scratch is one of the most difficult tasks because the input for this task is "sloppy". Therefore methods for these tasks are not very rigorous. However, every analysis in practice starts with this task, so it is very important.

model transformation

Transformation of a model can be done for several reasons. One reason to transform a model into another one could be that for the second model better analysis tools or techniques are available. Another reason could be that the second model is better suited for construction of the real system. This is the case if the second model is regarded as the blue print for the real system. Consider for example an object model, made in our formalism, in which the real system should be constructed with a relational database management system. We can transform our first model into a second one that satisfies requirements of the database

management system. The reason why we do not start with the second model immediately, is that the first model is more concise, better understandable and may be better to analyze. Another example is an actor model where stores are shared by several processors. If we have to realize the system on a computer network without shared memory, we can transform our model into a model that has only *private* stores, i.e. stores used only by one processor.

Of course the systems engineer has to prove that the transformed model is enough "similar" to the original one. Here the similarity relations of part II are useful.

verification

The possibilities to verify properties of realistic models are limited. There are two kinds of properties that can be verified:

- internal consistency,
- requirements formulated for the system to be constructed.

Internal consistency means that there are no "logical" errors in the model, for instance that the syntax of the model is correct, that the used types are consistent or that the system does not have any deadlocks. For some of these questions methods are available. Internal consistency of a model is no guarantee that the model describes the system that the systems engineer or his principal has in mind! To make sure the model describes what we want, we have to verify the properties formulated in the requirements specification. Often these properties are written down informally and the systems engineer has to translate them into some formal language first, for instance some form of logic. This translation is also a source of errors. A good strategy is to start with an informal description, then a translation to a formal one and finally a translation of the formal one into an informal description. The two informal descriptions should be consistent which can be checked by non-specialists.

This kind of verification is a hot research topic and there are not many results yet that can be applied to practical cases. Therefore in many cases the systems engineer has to use experimentation by simulation (validation) with the model in order to obtain confidence instead of certainty: experiments can only give counter examples for a property, but no proof in case a property holds. However if a model passes many tests, we get confidence that can be quantified with probability theory.

validation

Methods are no algorithms, so they are no sequences of unambiguous instructions to make or analyze a model. The distinction between guidelines and techniques is not very sharp. One could say that a technique is more rigorous, or "closer" to an algorithm than a guideline. In a technique the final result as well as intermediate results are precisely specified, however not the way to go from one step to another. As examples of techniques, we mention techniques for:

- the representation of the history of a system in an object model that only considers the actual state of that system,

- transformation of an object model in our framework to a relational model,
- verifying an invariance property in an actor model.

Examples of guidelines are: **guidelines**

- hints to represent a practical situation as a model,
- a checklist to perform a modeling task, in fact a *systems development life cycle* may be considered as a checklist too,
- conventions to be followed during a modeling task, for instance conventions for notations to keep models readable, or conventions for the use of specific constructions to keep models "well-structured".

For a complex system we cannot make a complete actor model at once, we have to construct it in steps. There are three well-known guidelines to develop a model. They have traditional names, that we will use here:

- The *process oriented approach*. In this approach the steps are as follows: **process oriented**

  1. Design a hierarchical actor model up to the processor level. Neither the processor relations for the processors, nor the object classes for the places (including stores) are defined in this step in a formal way, but they are described informally. Give for all processors their properties and give, if necessary, constraints of the delays of produced tokens. This activity will be called *actor modeling* and we will call the result of this activity an *incomplete actor model*, because term "actor model" is used for a complete formal description of a system. In the information systems community the activity is called *process modeling*. We, however, use the term "process" for an element of the behavior of a system. Instead of "incomplete" actor model, we will use the term "actor model" in case no confusion is possible.
  2. Design an object model for the total system. In this object model all complex classes that play a role within the system or in the communication with the environment have to be defined. The object model should include all relevant constraints, either expressed graphically or in predicate calculus and natural language. This activity is usually called *data modeling*. However, we will use the term *object modeling*, because we are not modeling information objects only, but also physical and conceptual objects. Complex classes are defined as in chapter 9, so no sophisticated value type is defined here.
  3. Assign complex classes to places. These first three steps are called *modeling*.

4. Specify for each complex class a value type.
5. Specify for each processor a processor relation. The last two steps are called *specification*.

data oriented

- The *data oriented approach.* Here the same steps are performed, but in a different order:

  1. Design an object model, including constraints.
  2. Design an incomplete actor model in the sense explained above.
  3. Assign complex classes to the places.
  4. Specify value types for complex classes.
  5. Specify processor relations.

object oriented

- The *object oriented approach.* This approach is different from the preceding ones in the sense that the three steps are performed simultaneously, however for each complex class separately. So the modeling task is not divided according to the different aspects of the actor model, but according to the different complex classes. Note that in the terminology of the object oriented approach, the concept of an object is used differently: a token is there an actor or a combination of an actor and tokens.

In the following chapters we first consider methods for making and transforming models. We distinguish methods for:

- actor modeling,
- object modeling,
- specification of value types and processor relations, with as final result a complete actor model,
- object oriented modeling.

We often mention properties of a (type of) model, but we do not give proofs here, because we concentrate on modeling issues first.

In this part we consider methods for construction of models and in the next part methods for analyzing models.

Chapter 12

Actor modeling

Actor modeling is the activity of making an incomplete actor model. In this chapter we will only consider incomplete actor models and therefore we use the term "actor model" as an abbreviation. Remember that the possible processor characteristics are *totality*, *functionality* and *completeness*. Further the *timing* of tokens is considered in the process model, if this is relevant. Note that we do not consider verification of properties of models in this chapter.

We distinguish making an actor model after reality, i.e. without any formal description to start with, and transforming a given actor model into another one which has structural properties the first one does not have.

12.1 Making an actor model after reality

There are two cases: the systems engineer has to make a model of an existing system, or he has to model a system he has (partly) in mind. In the first case he can observe the existing system and check if his model has the same properties as the real system, in the second case he can only check whether the system he has in mind, fits in the environment in which it should operate. From a modeling point of view these cases are not different: only the source of information is different.

We will consider several modeling problems. To illustrate these problems we regard two examples: order processing in the sales department of a company that delivers items from stock to customers, and traffic control at a railroad station.

order processing example

In the order processing system customers send a request a to the sales department to make an offer for delivery of items. (The characters refer to places in figure 12.1). Then the sales department produces an offer b, taking into account the inventory information which is shared with the environments. Now the customer decides to send an order c, which starts several activities in the sales department, and which results in an order confirmation d. Now the sales department sends a delivery order g to the distribution department. If the distribution department is ready to deliver it sends a message to the sales department

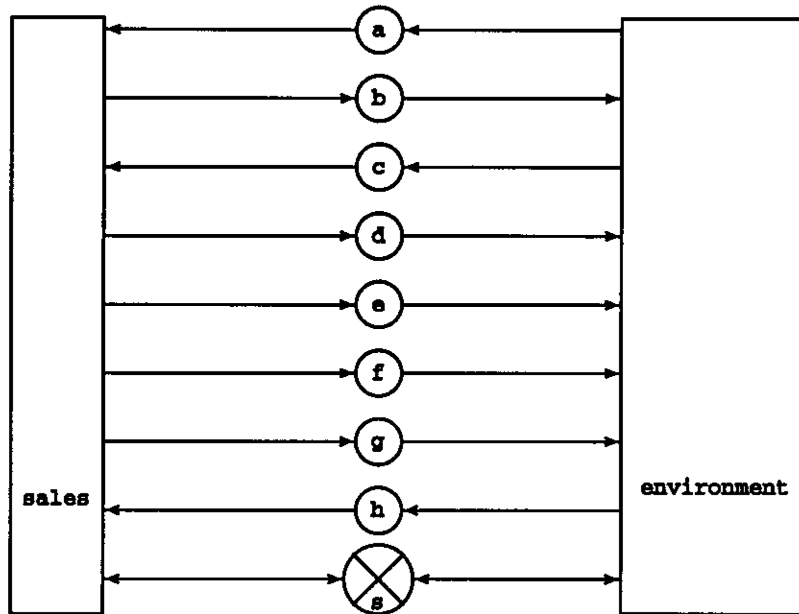

Figure 12.1: Order processing, context diagram.

h and the sales department notifies the customer and sends an invoice *e*. At the same time the sales department sends a copy of the invoice *f* to the finance department, which ends this *transaction* for the sales department. Of course there might be a lot of communication between the customer and the finance department concerning the payment of the invoice but that does not bother us here. We also neglect that the customer might be unhappy with the received items and that he contacts the sales department again about this order.

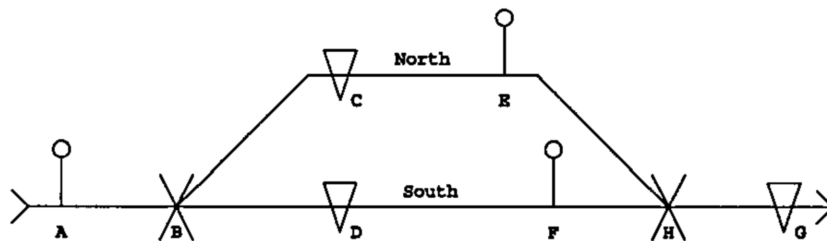

Figure 12.2: Railroad station, physical lay out.

railroad station example

The railroad station has a layout displayed in figure 12.2. Trains are riding in one direction (from left to right). One track is divided into two tracks at the entrance of the station. It is possible to load and unload passengers from both tracks at the platform. It is possible that one train is passing another one at the station. This system needs an information system that controls the use of the tracks at the station safely. The station master may decide when a train is able to leave the station, but the information system should not allow the station

master to give two trains a green light at the same time. Sensors along the railroad produce information objects that are sent to the railroad station to notify that a train has passed the sensor. Sensors C, D and G produce an information object in case a train has completely passed one of these points. Further there are semaphores along the tracks: A, E and F. If a semaphore is red a train has to wait, if it is green a train may pass. There is one switch B before the station and one join H after the station. Only B is important because it can be switched, H is always pushed into the right position by the trains.

actor modeling steps

Using the above examples, we will now show how to make an actor model of a discrete dynamic system. We always consider systems at the highest level closed; *top* is the actor at the highest level. The first step is to *decompose* (split, refine) *top* into a network of two actors, one representing the discrete dynamic system we are interested in and another one representing the environment. The latter is called a *context actor*. So the first question to be answered is: what is the boundary of the system we are studying? To answer this question we have to define the places through which the system of interest communicates with the context actor.

step 1: decompose top

context actor

In figure 12.1 we see the first decomposition of *top* for the order processing example. The only interesting thing on this diagram is the communication with the environment, here by means of four input places, four output places and one store s. We often decompose the context actor once because it allows us to structure the communication with the environment. In figure 12.3 we have decomposed it. Here we see some new actors: *finance*, *customers* and *distribution*. They belong to the environment, we call them also context actors. We left out all communication between the context actors because it is not interesting for us.

step 2: decompose discrete dynamic system

The second step, i.e. the first decomposition of the discrete dynamic system we study, (often) gives two actors: one representing the target system and one representing its information system. The *target system* is the system for which the information system works.

target system

information system

context diagram

An actor model showing the target system and the information system as one actor each and some context actors as well, is called a *context diagram*, so figure 12.1 and figure 12.3 are both context diagrams. In practice we start immediately with one of these diagrams.

If we want to *simulate* the system we study, we have to simulate the environment as well. However, it is impossible to do this exactly because we should specify the environment in detail. Instead of doing this we make models of the context actors as if they would operate in isolation. This means that we design a consumption/production behavior for them that is an *approximation* of the behavior of the real environment in the sense that the context actors are behaving like the real environment. Sometimes we may model a context actor by means of a *random generator*, which may imply that the context actor behaves more unpredictable than the real environment. This means that we test the system under circumstances that are more difficult than the real

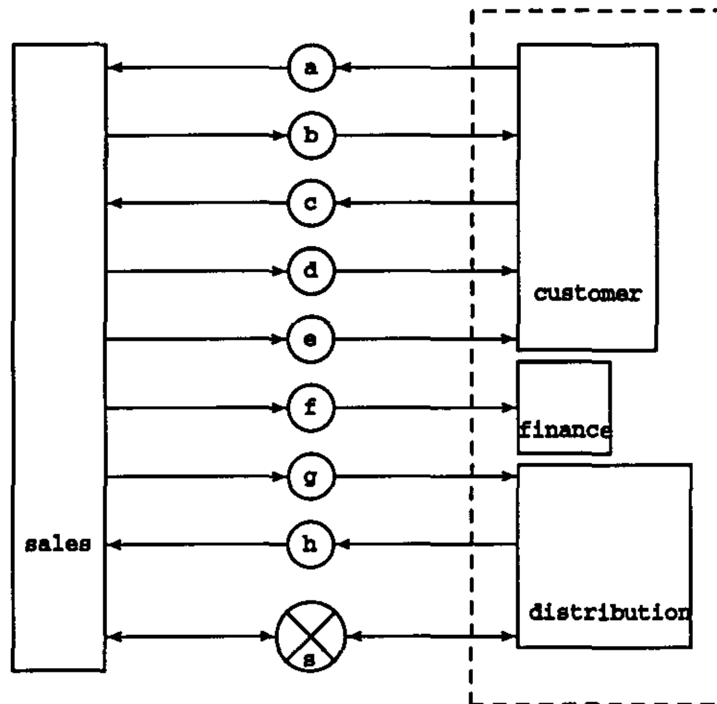

Figure 12.3: Order processing, first decomposition of the environment.

system will encounter.

Often we are modeling only an information system, as in the order processing example. In this case the system we want to model is an information system, exchanging information objects only with the context actors. Actor *customers* represents all customers of the company; only the exchange of information objects with the sales department is relevant. Of course, we could have modeled all customers separately by a context actor, but this would make the model vulnerable: if there would be a change in the customers population, our model would not be correct any more. (How the customers are modeled by one actor is discussed later.)

In the railroad example the object of study is a discrete dynamic system involving physical systems (the trains, the tracks, the semaphores). We distinguish the information system and the target system (which is a discrete dynamic system itself). The information system has two characteristic properties: all tokens are information objects and many tokens in the target system have a "counter part" in the information system, since the information system maintains an image of the state or history of the target system. The context diagram for the railroad example is displayed in figure 12.4. Here we see three actors: the *station*, which is the target system, the *traincontrol*, which is the information system and a context actor called *trains*. Context actor *trains* has a simple behavior: it will produce trains for place k and it will consume trains from place l (trains will have a direction). So there remain two actors

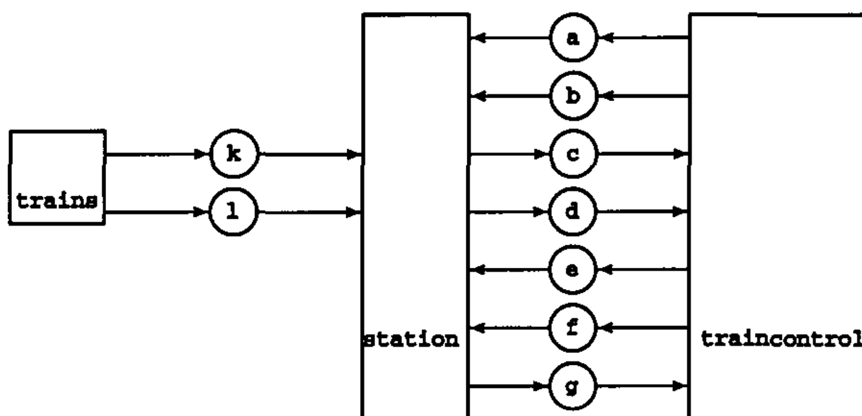

Figure 12.4: Railroad station, context diagram.

that we will detail to some extent. Channel a gives control tokens to semaphore A, b to the switch B, c and d pass sensor information from C and D respectively, e and f give control tokens to F and G respectively and by g the sensor information of G is passed.

Besides the target system, its information system and the context actors we often introduce some other actors in the context diagram: *measurement actors*. These actors are used to obtain statistical information from the simulation of the behavior of the other actors. So these measurement actors only consume tokens from the other actors and they have no influence on their behavior. Often a systems engineer has a "library" of measurement actors at his disposal with which he can collect information of the actors he studies. Note that the measurement actors only play a role in the model and not in the real system.

measurement actors

Summarizing we may find the following actors in a context diagram:

actors in context diagram

- target system,
- information system,
- context actors,
- measurement actors.

In the next modeling steps the target system and the information system are further decomposed. It is sensible to start with the target system, because the information system has to fit to it. In case the information system is our focus point, we do not model the target system into much detail. It is difficult to give a general rule how far to go. Often we encounter actors we are not able to describe completely. For instance, we may have to model human beings. This is, at least for systems engineers, an impossible task. Human beings may also be part of the information system if we decompose it. This kind of actors is not further decomposed. We regard them as *black boxes* and we call them context actors as well. So if we need their external behavior we replace

step 3...: stepwise refinement

them by some simple actor with a behavior that is at least as "rich" as that of the black box actor.

In general it is not wise to have many levels of decomposition because this makes a system difficult to understand. Ten levels seems to be a maximum while two to five levels seems to be "normal". There is also a guideline for the size of the network into which we decompose an actor: this should not contain more than ten actors, while five is a good size. (Combining these two limits would imply that models with a few hundred processors are "normal" and that models should never be larger than 10^{10}!) Making a model is more an *art* than trade, there are no "hard" techniques. Some guidelines to do the decomposition are:

decomposition guidelines

- in case of modeling an existing system, use the functional decomposition that is already there, for instance the partitioning in departments, business units or task forces;
- if the system is delivering different products or services, follow their path through the system to find tasks that can be performed by one actor;
- do not consider the task of an actor in detail, but only its input/output behavior; draw a "boundary line" and see what kind of tokens are passing the line;
- let the task of an actor be easy to understand and to describe in natural language; if the task is too complex it might be divided into several tasks, in this case the so called *cohesion* of the actor was too low;
- avoid many connections between two actors, which is called the *coupling* of actors; if there are many connections the partitioning of tasks over the actors might probably be improved;
- often it is useful to work bottom-up: first make a detailed model and then cluster actors into higher level actors.

When we are refining actors we often discover that we made errors at higher levels in the actor model, mostly because we forgot some communication between actors or because we have considered different types of tokens as one type. This is not a problem although systems engineers hate to modify already made diagrams. However, it is very important to keep the diagrams consistent, which is often called *balanced* and which means that the diagrams together form one hierarchical actor model.

end of decomposition

At a certain point the actors become *elementary*, which means that they are processors and will not be decomposed further. A good check to see if an actor can be considered as elementary is that it should operate *memoryless* and it should be able to perform its operation(s) in one event. Hence the tokens it will produce may only depend on the tokens consumed and we must be able to consider its operation as one

transition of the system. Note that it is often possible to consider the stepwise processing of a task as one event. For instance, the execution of a computer program can be described as the evaluation of one function application. In a functional model it is sufficient to represent such a program by one processor. If we want to make a construction model it is necessary to model a program by a processor network in which the processors correspond to elementary program constructs. In actor modeling we do not determine types of tokens and the functionality of processors in detail, we only specify the processor characteristics. In case a processor relation is total and complete (i.e. input and output complete) the actor behaves as a classical Petri net (cf. chapter 5) if we discard the values of the tokens. So, even without completing our model we are able to do some analysis. In many cases it is possible to model systems as classical Petri nets, but these networks become very large and confusing. So we should avoid refining too far and stop at a level where we discover memoryless actors that perform operations in one event.

For the two examples we considered above, we will give a decomposition to the processor level. For each processor we will determine the processor characteristics.

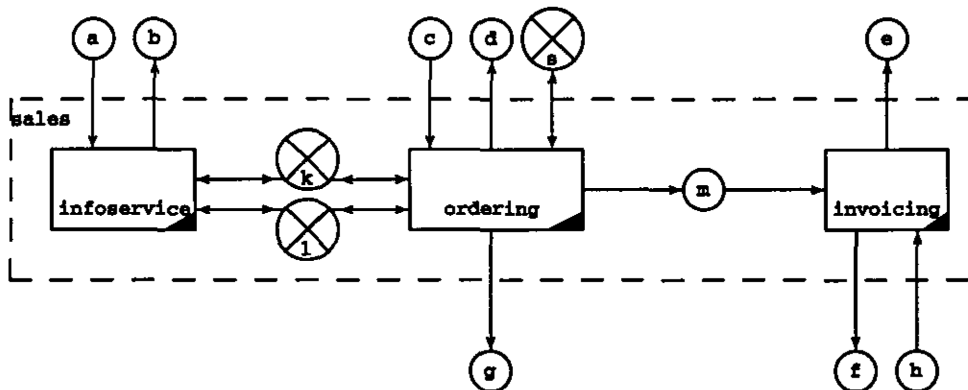

Figure 12.5: Order processing, "final decomposition" of *sales*.

order processing (continued)

First we consider the order processing example. In figure 12.5 we show the decomposition of the sales actor. Here we see processors only, so we are already at the lowest level of decomposition. Processor *infoservice* answers questions a of customers by sending them an offer b. Processor *ordering* consumes orders c and produces a delivery order g for the distribution department, a confirmation d for the customer and the necessary data for delivery and invoicing. This is an information object sent to place m. Processor *ordering* uses also the inventory information s (which is shared with the distribution department), the customer information in store l and the price list in store k. In fact *ordering* is updating s and l as well as retrieving information from them. Processor *invoicing* is triggered by a delivery notice h from the distribution department saying when the ordered products will be

delivered. It selects the right order information object from m in the same event. So *invoicing* will not be total, since it is only able to process combinations of tokens from h and m for which the order identifications match. Note that the distribution department may have very little information about the order, for instance it may not know the prices the products are delivered for or who is paying for them. Processor *invoicing* also produces a booking for the finance department. All processors are complete; *invoicing* is functional and *infoservice* too. However, *ordering* may be non-deterministic and therefore non-functional. This is the case if there is some flexibility in the determination of volume reductions.

This is our functional model of the sales department. Note that it is not known yet if persons play a role in this system. For instance, if the communication with the context actors is realized by means of *electronic data interchange* there is no need for persons. However in todays practice persons will be involved. Maybe each processor is realized by several persons or all the processors by one person. Here we are only interested in a functional model. If we consider a construction model of this system it should be consistent with this functional model. Note that there are several ways to decompose the higher level of the actor model. The decomposition is adding information. The original informal description of the system was not detailed enough to validate the model. The model is what the systems engineer thinks the system is. Finally we remark that this model is far from complete, for instance there are no facilities modeled to update the price list.

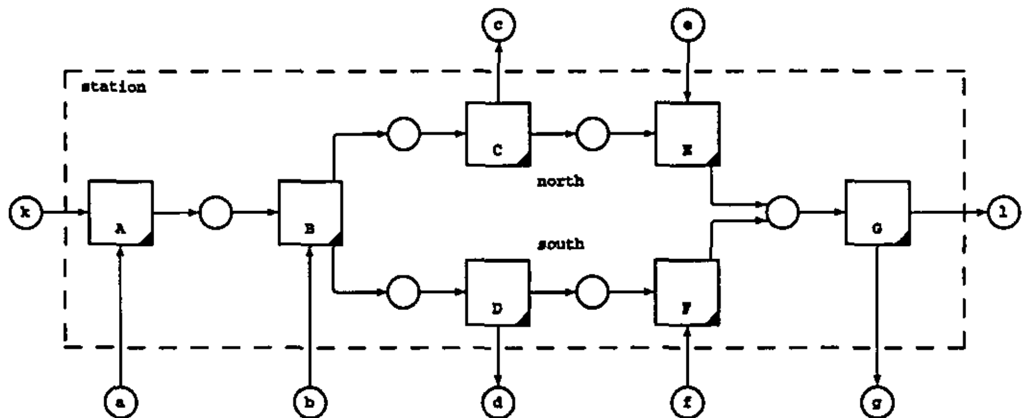

Figure 12.6: Railroad station, functional model of actor *station*.

railroad station (continued)

Now we will look at the railroad station. In figure 12.6 we give a functional model of the target system, the station. Here we see that the semaphores (A, E, F), switch (B) and sensors (C, D, G) are modeled as processors. The join H appears as a place n. As we have seen in figure 12.4 the places a through g model the exchange of information between station and train control. Of these places, only place b has a type that has at least two values, namely the strings "north" and "south", while the values of the others are not relevant (only the fact that there is

a token is relevant here). The other places (k, l, n and the unnamed places) are used by trains. So their type is "train". Note that the processors consume or produce physical objects as well as information objects. This often occurs in models, because transformations or translations of physical objects require information. Only the switch processor B will use the value of the token of b to determine if the switch should go to the north track or to the south track. Only processor B has incomplete output: it allows a train to go to only one of the tracks. Furthermore B is total, input complete and functional. All other processors are very simple here: they have all the processor characteristics and their function is the identity function for all the places of the train-type, so they reproduce the train they consume. In the initial state there are only tokens in k.

Next we consider the functional model of the information system of the target system: the train control, see figure 12.7. This model looks

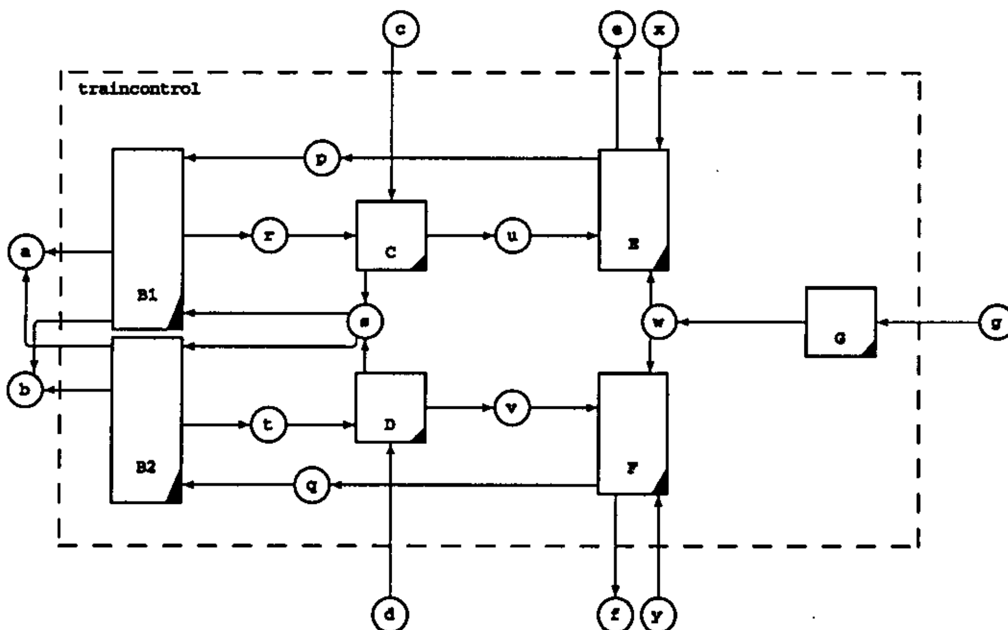

Figure 12.7: Railroad station, actor *traincontrol*.

very similar to the model of the target system! Places with the same names in figure 12.6 and figure 12.7 are supposed to be the same.

This is often the case: the information system should maintain all relevant details of the state of the target system, so from the state of the information system we should be able to derive the state of the target system.

Now we will study the model of figure 12.7 in more detail. The names of the processors in the model correspond to the names of the processors in the model of the target system. Only processor B of the target system, the switch, is divided into two processors here: B_1 and B_2. That was not necessary but it makes it possible to model the

information system as a classical Petri net here: all processors satisfy the three properties and all token values are irrelevant except for the tokens produced by B_1 and B_2, that give their token the value "north" and "south" respectively. (This can be done in many cases.) We also see the places for the exchange of information objects with the target system. The places x and y are used by the station master, so they have to be connected to a context actor representing this person. The places p and q are used for *feed back*, only if processor E has fired, processor B_1 can be enabled again to allow a train into the north track.

We assume that in the initial state there are tokens in s, p, q, w and k only. In k the number is arbitrary and in the other places the number is equal to one. This means that B_1 and B_2 are enabled and that there are no trains in the station. It is decided by the Demon which track will be opened for a new train, i.e. whether B_1 or B_2 will fire. It is easy to modify the model such that the station master can make this decision. (This is an exercise.)

place invariant

It is easy to see that the amount of tokens in the places p, r and u together is equal to one in any reachable state. The same applies to t, v and q. Such a property is called an *invariant*, to be precise a *place invariant*.

If there is a token in p then the north track is free and if there is a token in r then a train is entering the north track and it has not yet passed the switch completely (i.e. the train has past sensor C). If there is a token in u there is a train at platform north and switch B can be used by a train that goes to the south track. In this case the invariants are easy to verify. In part IV we will see a method to find and prove them. Another invariant is that places r, s and t have exactly one token in total (if the initial state has this property), which means that at most one of the processors B_1 and B_2 is enabled in each event. If there is a token in r then the switch is open for a train to go into the north track and only after this train passes the sensor C there will be a token in s again which allows to set the switch to the south track (if this track is free, which is indicated by the existence of a token in q). Note that B_1 and B_2 enable the switch and the semaphore A in the same event, when it is safe to enter the station.

The station master may decide which train will leave the station first by putting a token in x or y. If he erroneously puts a token both in x and y no harm has been done: there is at most one token in w, so E and F cannot fire both. Furthermore, the semaphores E and F are only enabled if sensor G has given a signal which means that the preceding train has passed the intersection, i.e. switch H. To prove that the whole system works correctly requires more arguments, we do give them in part IV. However, it is easy to simulate the system by hand to get a feeling for it and to detect mistakes.

Note that there are other ways to represent the state of the target system: for instance with one store containing one complex object, as displayed in figure 12.8. Here the store i represents the contents of the places p, r, u, w, t, v, q and s of figure 12.7. The processors P, Q,

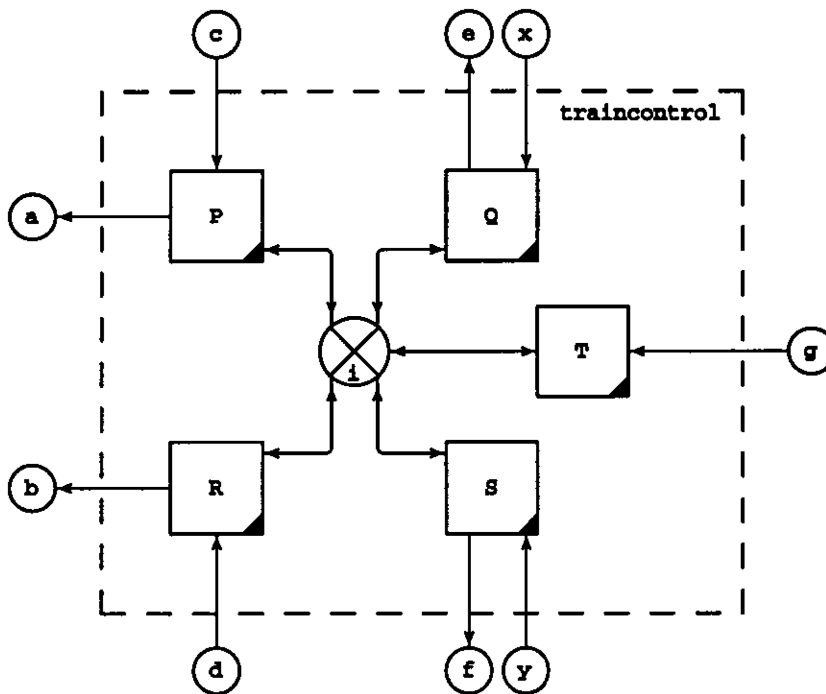

Figure 12.8: Train control with a store.

R, S and T of figure 12.8 are only triggered by external places, i.e. by the sensors of the railroad or by the station master. Although this actor model is much more simple than the one displayed in figure 12.7, the processors are more complicated here, and the verification of the correctness is more difficult.

token time

In the examples so far we have not considered the timing of tokens. In many cases this aspect is also important in an early stage. For instance there could be an actor representing a clock, that creates *triggers* (tokens) for processors that have to be activated at particular times. In general it is "dangerous" if the time aspects of the tokens play an essential role in an information system, i.e. if the functionality of an information system depends on the processing speed of its components. In real time systems this is sometimes the case. There is a trend in designing information systems that are *delay insensitive* which means that their functionality is independent of processing speed.

more guidelines

We conclude with some other guidelines:

- give actors a name that reflects *action*, for instance: "updating" or "painting", while places get names representing a state of *rest*, for instance "waiting for shipment";
- in case there is only one connection from a place to an actor, we may use the name of the place for the connector;
- it makes sense to give actors a number in such a way that their descent can be traced;

- it is always important to have an informal description of a model in every stage of development and this description should be updated after further refinement or formalization; always list the assumptions made;
- check whether in a model every place has at least one input and one output actor; if not this place might become empty or the number of tokens in it might grow unlimited, which is seldom what we want;
- check if actors have input and output connectors, if not it are probably context actors;
- in refining a model only use stores that are shared by more than one actor, only on the lowest level, where we have only processors, we use stores that are private for one processor;
- try to make simple couplings between actors, i.e. the knowledge one actor has to have of another one with which it communicates, is as little as possible;
- make as much as possible *reusable* actors, by keeping interfaces simple and by using polymorphic functions and type variables in the specifications of places and processors (cf. chapter 6).

12.2 Characteristic modeling problems

We will now consider several problems that occur in many modeling situations. We will illustrate these problems as much as possible with the two examples given above.

1. The role of objects and actors.

object roles

Objects or tokens can play different roles in a model. They may represent:

- *physical objects*, like the trains;
- *clusters* of physical objects, like a set of cars in a garage;
- *messages*, like the order confirmations;
- *databases*, i.e. a complex object that represents (a part of) a state of a system, like the customer file; often these objects reside in a store, but not necessarily;
- *stage indicators* that represent the stage or state of some "entity", like the objects (tokens) in the places of the train control system (see figure 12.7); in these cases the places may contain at most one token and its type is irrelevant;

- *signals* like the signals exchanged by the station and the train control system; only the processors B_1 and B_2 produce messages and all other interface places contain signals; a signal is like a stage indicator, its value is irrelevant but there may be several signals in one place.

The role of actors and processors in particular, is one of the following:

actor roles

- *complete systems* that consume and produce tokens, with or without memory and *self-triggering* (self-triggering means that the processors produce their own input tokens); in particular human beings can be modeled as actors;
- *transformers* or *transporters* of tokens;
- *markers of activities* in networks where places represent stages and tokens stage indicators.

It is important to indicate in an early stage of the development of a model what the role of an actor, a token (type) or the place it resides is. This may already determine properties of the actor model that can be verified at later stages in the development.

2. Modeling an entity as a token or as an actor.

token or actor

Often we have the choice to model a real-world entity as a token or as an actor. We start with an example.

In a production system it is quite natural to model the products as tokens, but for resources this is not so clear. Consider for instance a simple production system in which a product is made in one production step from raw material to the final product by one machine as illustrated in figure 12.9. Here A denotes a context actor that produces raw material and C a context actor that consumes final products, while B represents the machine. It is clear that the quantities of raw material needed for one product are modeled as tokens and the products too. The machine is one processor here. A disadvantage of modeling machines as actors is that the model is vulnerable for changes in the number of machines, like we have seen in the order processing example above with the modeling of customers. Consider the same production system but with three machines instead of one, as displayed in figure 12.10. In figure 12.11 we find a model with arbitrary many machines. Note that B now represents a *machine operation* instead of a machine. In place q the machines reside. Now the machines are tokens too. This model is only interesting if the manufacturing of products takes time since otherwise there is not much difference between this model and the first one.

Now we modify the problem a little: we assume that there are two types of operations, B and E and that the production process first requires the use of a machine of type B, then one of type E and finally one of type B again. We can model this in two ways: there is still only one actor for each machine operation (see figure 12.12) or there is an

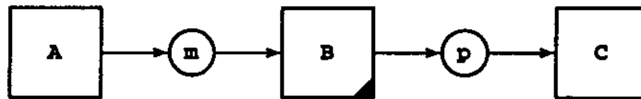

Figure 12.9: A simple production system.

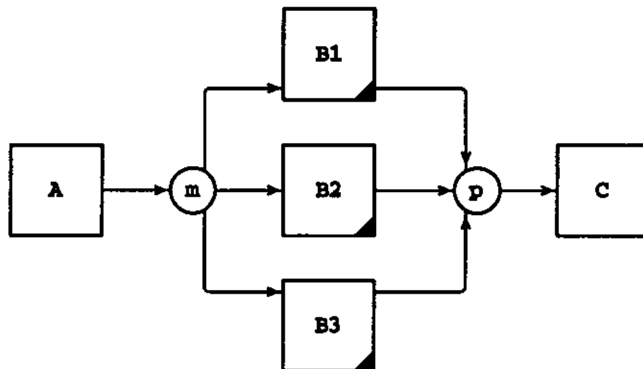

Figure 12.10: The same production system with three machines.

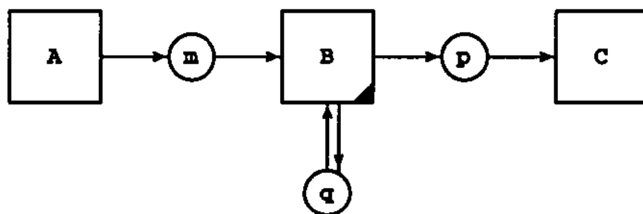

Figure 12.11: The same production system with arbitrary many machines.

actor for each step in the production process (see figure 12.13). In the model of figure 12.12 processors B and E must decide if the produced token is ready or that it needs another production step. So the tokens should contain this information. In the model of figure 12.13 the tokens do not need to contain any processing information at all. Note that in these models places q_1 and q_2 represent the machines of type B and E respectively. It is always a trade off between a complex network with simple object types and simple processor relations for processors, and a simple network with complex object types and complex processor relations.

The construction presented here can be applied in general. If a processor represents an entity that operates on objects, then we can modify the model such that the processor becomes the *activity* of the entity instead of the resource itself. Then we have to add a place that is connected bidirectional to the processor and which contains a token that represents the entity. If we do this systematically, we obtain a model in which all entities that live for a time period are represented as token and in which all processors represent instantaneous activities.

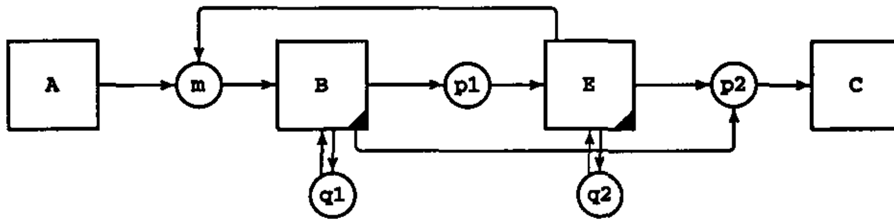

Figure 12.12: One processor per machine operation.

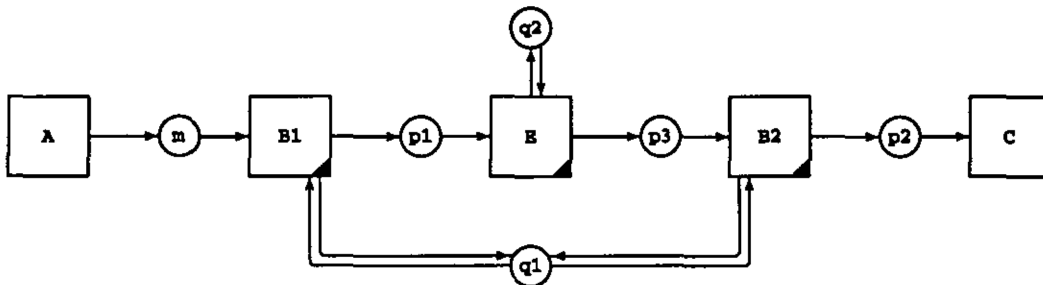

Figure 12.13: One processor per type of production step.

3. Modeling a file as one token or a set of tokens.
Consider a file of items. The same model can be used for a warehouse with one type of physical objects. We can model this file as a place containing a token for every item in the file, or we use a store with one complex object representing the whole file. Often we need to inspect all the items in the file and this is quite difficult due to the fact that processors have no means to check whether a place is empty or not.

file as one token

In figure 12.14 we see a simple file management system. Processor p adds a token of type $\mathbb{N}$ from place a to the file stored as one token in store k. The type of this store is $\mathbb{N}^*$. Processor q inspects this store and computes the sum of the values of the items, whenever it receives a signal in place b. The processor relation for both processors is quite

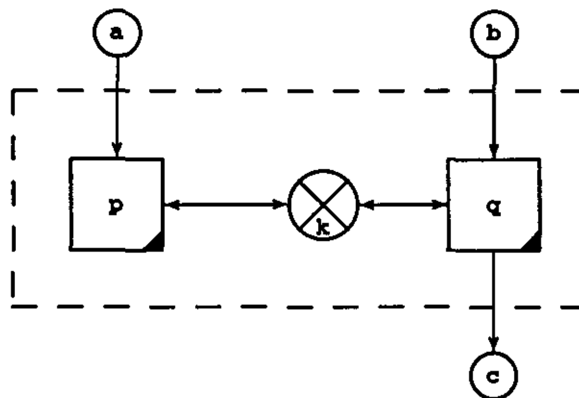

Figure 12.14: File maintenance with a store.

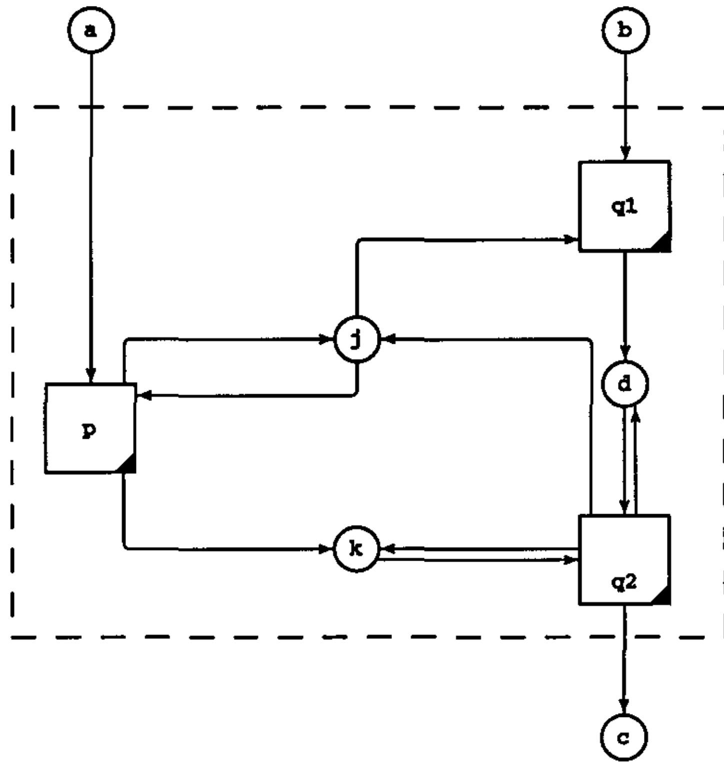

Figure 12.15: File maintenance without a store (1).

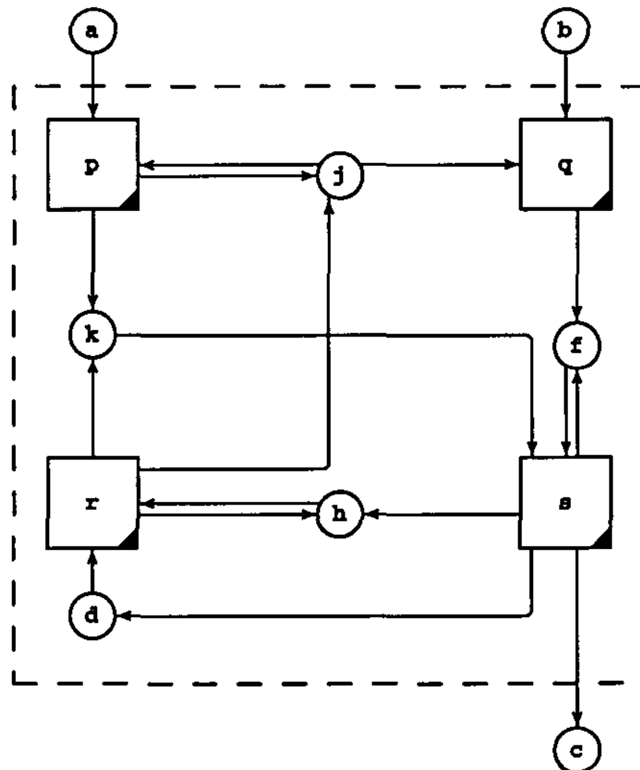

Figure 12.16: File maintenance without a store (2).

easy.

However, if we want to avoid the use of a store to model the file, we may replace the store by a channel where each "stored" item is modeled by a token. Using preconditions we may select the right token for an operation. However, if we want to perform an operation on *all* items, we need a more complicated solution, like the one displayed in figure 12.15. We assume that each query token on b has as value some unique number Q. Store k has been replaced by a place k containing a token for each "stored" item. Place j has a token in the initial state with a value indicating the number of items in the file (i.e. tokens in place k). Each token in k has a value that represents the stored item and the stage in the inspection process ("inspected by query Q", where Q denotes the last query in progress). Processor q_1 consumes a token from place b and the token from j and produces a token for d. The value of the token in d is a row (m, n, o, z), where m denotes the total amount of items, n the amount of items already inspected, o their sum and z the query identification (Q). Then processor q_2 consumes all tokens from place k and puts them back with status changed into "inspected by query Q". When all tokens have been treated the answer is produced at place c and the token for place j is restored.

file as set of tokens (1)

If we do not want to add processing information to the "stored" items we need an even more complicated solution like displayed in figure 12.16. Again place k contains the file in the initial state as individual tokens, one for each item. Also place j has a token in the initial state with a value indicating the number of items in the file. No other place has a token in the initial state. Processors p and q are complete, while s and r are only input complete. All processors are total and functional. As long as no request appears in place b, processor p may add items to k, while updating the token in j. When a request arrives in b a rather complex process is started. First the token of j is transferred to f and then processor s transfers all items from k to d, while inspecting their values and adding these values in the token in f. The value of the token in f is a row (m, n, o), where m is total amount of items, n the number of items seen so far and o is their sum. As soon as the items are counted, i.e. m equals n, processor s produces the accumulated file value in place c and it puts a token in place h with the number of items in the file as value (which is known from g). Further it does not return the token in f as was done in the preceding firings. Now processor r starts and carries out a process similar to s: it returns the items to place e, while counting the items seen so far and it ends with the return of a token to place j.

file as set of tokens (2)

This example shows that it is very convenient to use a store to represent a file and that rather complicated constructions are needed otherwise.

4. Knowledge in a processor or in a store.

processor or store

The processor relation of a processor often uses constants, such as a number or a finite binary relation. Consider for example a processor

that produces the amount of taxes to be paid when an income tax return is "consumed". The tax table will be used. The question is, should we put this tax table in the processor specification or should we define a store from which the processor reads the table. If there will be no updates of the tax table it makes sense to put the table in the processor specification, because it simplifies the actor model. However, if updates of the tax table are possible we have to introduce a store and also an actor (maybe a context actor) that updates the store. It is useful to check whether stores in an actor model are updated or not. If not there is probably something wrong: either the updating actor is missing or the store could be incorporated in all the processors that are using the constant. Note that in practice very little constants occur, in most cases the "constants" turn out to be variables. However, we may consider them as constants: changing them would then be understood as a change of the system, which is called *second order dynamics*.

5. Direct addressing or broadcasting.

We try to make models as modular as possible in order to be able to *reuse* parts of them for other models or to be able to adapt a model easily. Therefore actors have connectors that can be attached to places in a later stage, without changing the actor itself. If we have an actor that sends messages to other actors, in fact to their input places, then we have to know precisely which actors will get messages, in order to specify the connectors of the sending actor. This might be cumbersome because we do not know the number of addressees yet, or the number might change in the future. This solution is called *direct addressing*.

direct addressing

A better solution is to send all the messages to one place (so we just need one output connector in the actor) and only let the receiving actors consume messages that are addressed to them. Of course this requires the use of a precondition in the receiving actors. This solution is called *broadcasting*. The messages have to carry an address and each receiving actor has to know its own address, which is not the case if we use direct addressing.

broadcasting

In figure 12.17 the two cases are displayed: left direct addressing, right broadcasting. Actor p is the sender and q, r and s are the receivers.

In the second solution the sender still has to know all addressees. If we want a more realistic model of broadcasting the sender should not have to know the addressees nor their amount. There are two possible solutions to this problem: either add an actor between the sender and receivers that performs the addressing task (i.e. split p), or use a solution like the one displayed in figure 12.18. In figure 12.18 a message in d is "read" (consumed) by a receiver and then put back. Each receiver has a precondition saying that he should not read the same message twice: for this the type of d is now enhanced with a set indicating which receivers have already read the message. Initially this set is empty. When the size of this set is equal to the number of receivers, the message is discarded by a special processor t.

"real" broadcasting

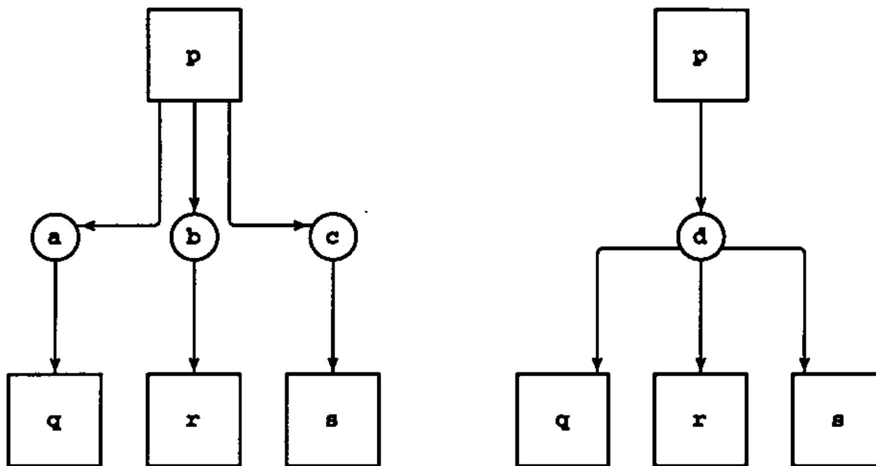

Figure 12.17: Direct addressing versus broadcasting.

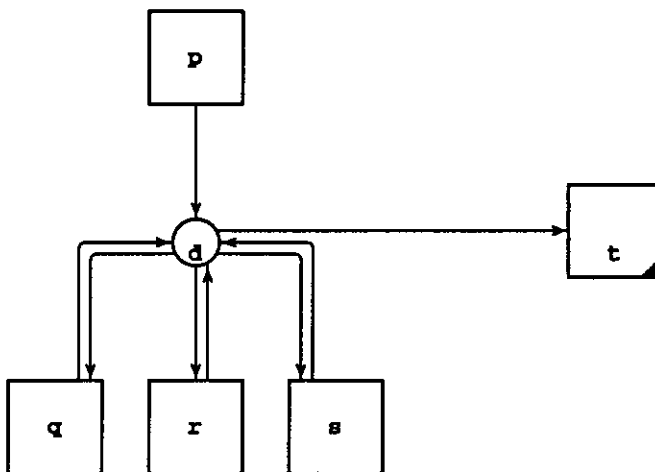

Figure 12.18: "Real" broadcasting.

6. Sequential processes.

A *sequential process*, like the execution of a computer program or a production process in a factory, can be modeled as a network in which every processor represents a processing step and every place a stage in the process. A characteristic of these networks is that every processor (except for the first and the last) has exactly one input place and one output place. (This is the property of a *state machine*, a kind of actor discussed later.) One of the characteristic features of an actor model representing a sequential process is the fact that the total number of tokens in the network is constant and equal to the number of processes that might be active simultaneously. This number is equal to the number of resources that can work simultaneously. (In traditional computer systems this number is equal to one.) In figure 12.19 a sequential process is displayed. We may interpret this example as a computer program

sequential process

state machine

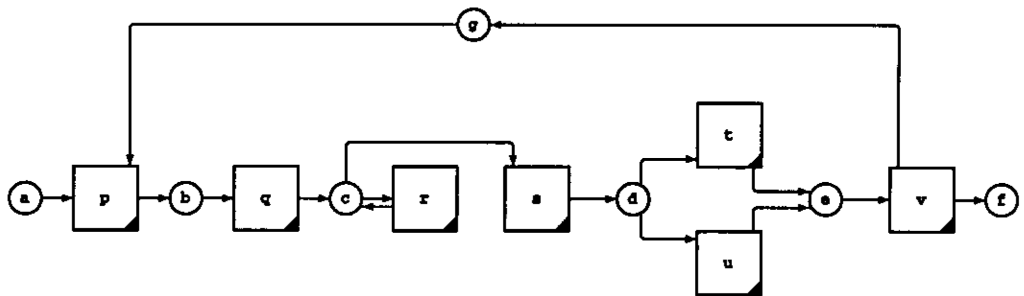

Figure 12.19: A sequential process.

with the well-known constructs: *selection*, *iteration* and *assignment*. In fact we may consider this actor model to be a *flow chart*. (Although flow charts are not used in software engineering any more, they still are used in other engineering disciplines). Processor p represents the buffer control of the process: only if places a and g have a token it may fire. The tokens in g denote the number of free resources that may perform a sequential process. Processors r and s represent an *iteration*. They have preconditions to determine if the iteration is ready or not. So only one of them will be enabled to consume the token in place c. Processors t and u form a selection. Their preconditions form the *if then else* construct. Processor q represents an assignment. All processors are complete and functional. Processors p, q and v are total. Processor v marks the end of the process and returns the resource to place g to allow processor p to start a new job. So the process itself is the network between processors p and v. In the initial state there are only tokens in g. We may replace all processors, except for p and v by an actor that satisfies the property that it has exactly one input and one output connector and that it will produce one token if and only if it has consumed one token.

object life cycle

Sequential processes can be used to model *object life cycles* in the object oriented modeling approach. Then there is for each complex class one sequential process. They can also be used to model *transaction processing*: for instance in a database system each transaction can be considered as an object having a life cycle.

7. Synchronization.

synchronization

Here we consider two sequential processes that may communicate. If one process needs the processing of another it will send its "job" to the other process and waits till its returns. This is called *synchronization* of sequential processes. In figure 12.20 we see two sequential processes. All processors are complete and total. In the first process actor b produces a token that needs further processing in the second process, which is also a sequential process. Processor g of this process waits till the token to be processed arrives in place p and the result is delivered in place q by processor h. The first process waits, which is expressed by the completeness of processor c. Since the two sequential processes have almost the same structure, we can give a similar description for the

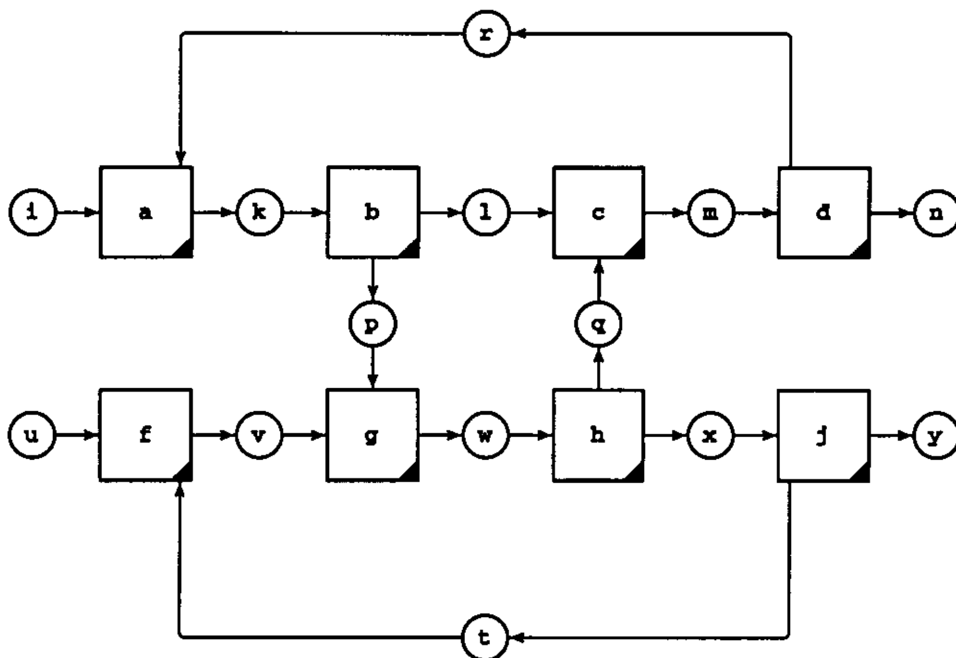

Figure 12.20: Synchronization.

second process. Note that processors b, c, g and h have exactly one input and output place within their sequential processes.

8. Mutual exclusion.

mutual exclusion

Mutual exclusion is a frequently occurring form of communication of processes, which share a resource that can be used by only one processor in one event. In the train control system (cf. figure 12.7), we already find such a situation: the token in place s represents the "tongue" of switch B that can be used for the north or the south track, but not for both in the same event. Another form of mutual exclusion is the use of a store by two processors: only one of them can use it in one event or during a time period. Here the network takes care of the problem! A more general case of mutual exclusion is displayed in figure 12.21. All processors are complete and total in this example. In the initial state there are tokens in r, t and s only. Here both sequential processes need the (single) token in place s. Processors b and g are waiting for this token and d and j return it after c or h have used it. Note there is no guarantee that a system that needs the resource will get it: the other system could take it always just before the first one. Using the time mechanism it is easy to obtain *fairness*, i.e. to guarantee that every request is satisfied. We only have to require that the tokens produced for r and t get a positive delay and that the resource has no delay in place s.

fairness

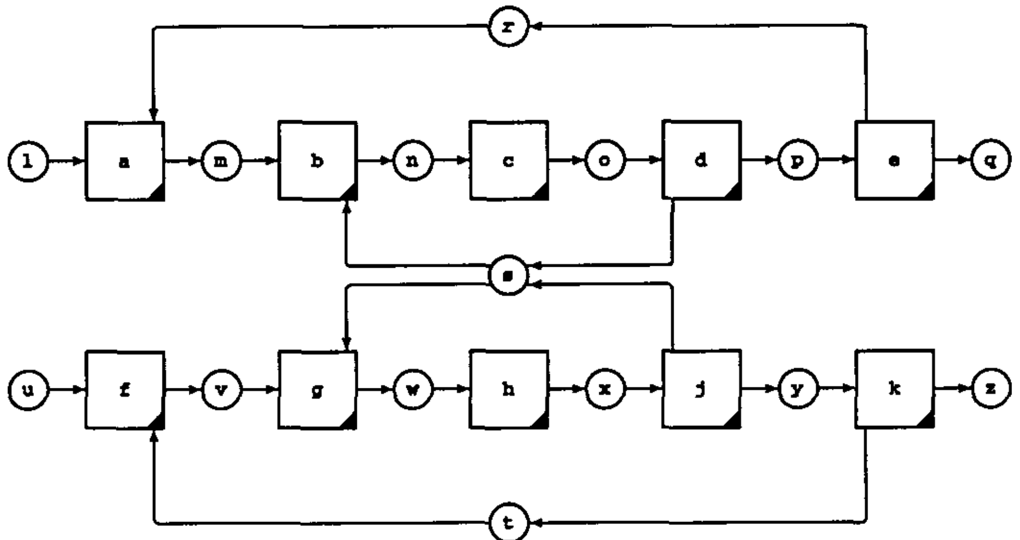

Figure 12.21: *Mutual exclusion.*

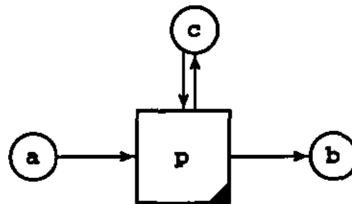

Figure 12.22: Modeling processing time (1).

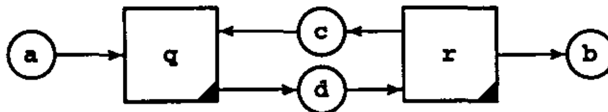

Figure 12.23: Modeling processing time (2).

9. Processing time and time-out.

processing time

We have already seen in part I how we may represent the fact that processors need time to perform their operations. In figure 12.22 and figure 12.23 we show two constructions. In the first one processor p produces a token for place b with a delay t which is equal to the delay of the produced token for place b. This delay represents the processing time. Although the token in b is immediately there, it is only available to other processors after the delay. In the second solution (figure 12.23) we see two processors q and r. Firing of q represents the start of the processing and the firing of r the end of it. Only the token in place d gets a delay equal to the processing time.

The processing time can be used to solve a problem we encountered before: when we modeled a file as a set of tokens, we found that it was impossible to test if a place was empty. So we had to use another

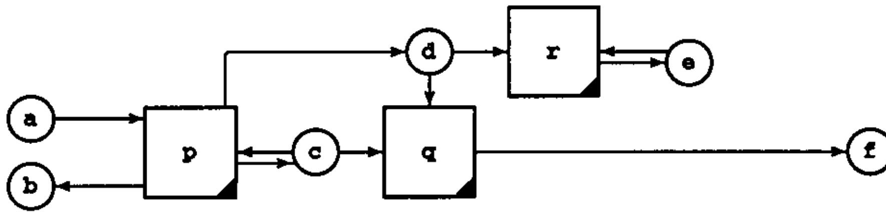

Figure 12.24: Inspecting a place using a time-out.

solution to see if we had inspected all tokens (see page 151 where we "stored" the amount of tokens in a separate place). With a *time-out* construction it is possible to determine whether a place is empty or not without counting the number of tokens in the place. A time-out is a way to decide whether a processor is enabled to fire or not, based on the time that has passed since its last firing. In figure 12.24 we consider a part of the file maintenance problem. All processors are total and input complete, only processor q is not output complete however the others are. The delays the processors give to their output are: tokens in c and e are delayed three time units, by processors p and r respectively and the tokens in d will be four time units behind. Channel a contains the file with items and after inspection the items are put into place b. The intermediate result of the inspection of the file is put into place c by processor p and the final result, if the whole file is inspected is put into place f by processor q. The initial state of this system is important: in a is the file, in c is a token with delay zero, in d a token with delay one, in e a token with delay two and the other places are empty. Processor p inspects an item every three time units, and processor q cannot take the token from place c before place a is empty because this token is "stolen" by processor p before q is enabled.

time-out

To verify this statement, we show by induction the existence of tokens in places c, d and e that become available at time points $3k$, $3k+1$ and $3k+2$ for $k \in \{0,1,2,...\}$ respectively, as long as place a is not empty. For $k = 0$ this is guaranteed by the initial state. Suppose the statement is true for k. Then processor p will fire at $3k$ and it will produce tokens for c and d that are available at $3(k+1)$ and $3(k+1)+1$ respectively. Note that there is still in d a token that is available at $3k+1$. At $3k+2$ processor r is enabled because of the token in e and it will consume the token in d and reproduce a token for place e that is available at $3(k+1)+2$. Now there is again only one token in d. So the statement holds for $k+1$. Only if processor p consumes the last token from a, say at time $3k$, then processor q will be enabled at $3k+4$, because there is a token available in c at $3k+3$ and in d at $3k+4$, while processor r has to wait till $3k+5$ to be enabled. So if q is enabled we know that a is empty, because the time for processor p has expired.

This example shows how powerful the time mechanism is, but one should be careful with it because realization of systems when the functionality of the model is using time aspects is difficult: systems become

real-time systems!

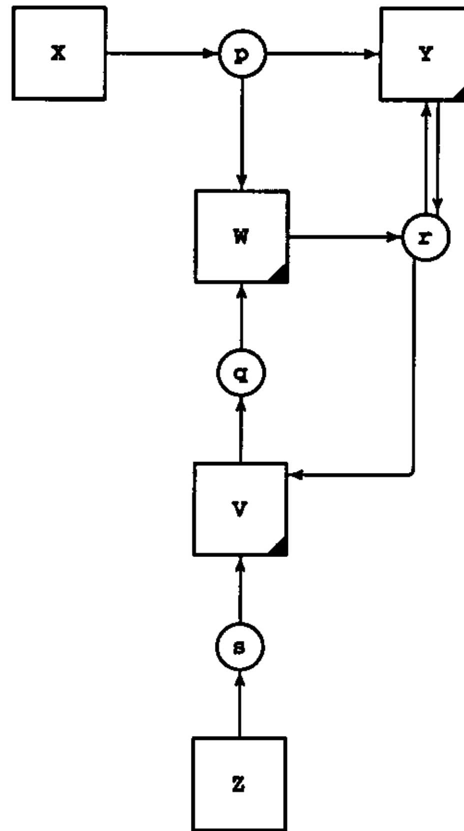

Figure 12.25: Token cancellation.

10. Token cancellation.

Sometimes the following situation occurs: a token is put into a place with a delay by some actor X to be consumed by some processor Y and before the delay has expired another actor Z wants to prevent that the token will be consumed by Y. Then Z sends a *cancellation* token that should activate some other processors to consume the token. However, these processors are of course not able to consume the token before the delay has expired. In figure 12.25 a solution is displayed. We introduced two new processors V and W and some extra places q, r and s. As soon as Z puts a token in s processor V consumes the (only) token from r and puts a token in q. Now Y is not able to execute because there is no token in r. (We assume that all processors are complete.) As soon as the delay of the token in p expires, W will consume the token in p and W will reset the system by putting a token in r. (So in the initial state there is one token in r and tokens in r will have no delay.) Note that we had to modify processor Y by connecting it to r. It is easy to modify the model in order to avoid modifications of Y. This construction only works for cases where Z wants to cancel an arbitrary token in p. In case Z is more selective in the sense that it will only cancel a token in p with

cancellation token

a particular value, then the construction has to be adapted. (This is an exercise.)

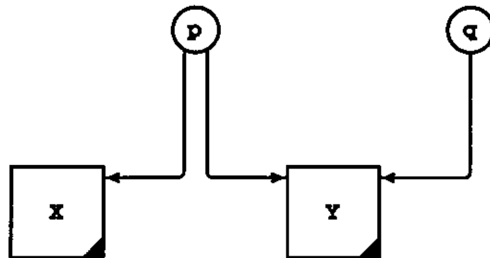

Figure 12.26: Token selection.

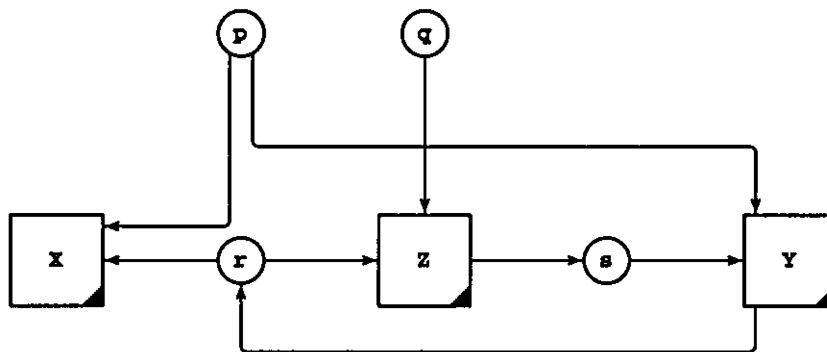

Figure 12.27: Priority in selection.

11. Priority in token selection.

token priority

Consider the situation displayed in figure 12.26. If there is one token in p and one in q such that the time stamp of the token in p is greater than the time stamp of the token in q, then the Demon will decide if X or Y will execute. If we want to give Y priority over X, we can modify the model as displayed in figure 12.27. All processors are complete. In the initial state there is one token in r. As soon as the token in q is available, Z will consume it, together with the token in r. Then X is not enabled if the token arrives in p. Processor Z also duplicates the token in q to a token in s. Now Y is enabled and after its execution there is again a token in r so that the system returned to its initial state. Note that if the tokens in p and q arrive at exactly the same time the Demon still decides between executing X or Y. (It is easy to modify the model such that X and Y do not have to be modified.)

12. Continuous processes.

continuous processes

Sometimes we want to model a process that is typically continuous. In part I we said that we restrict ourselves to discrete systems, however we are able to model some continuous systems as well. Consider for example the chemical process displayed in figure 12.28 we see the production of salt from $NaOH$ and HCl. We modeled every reservoir

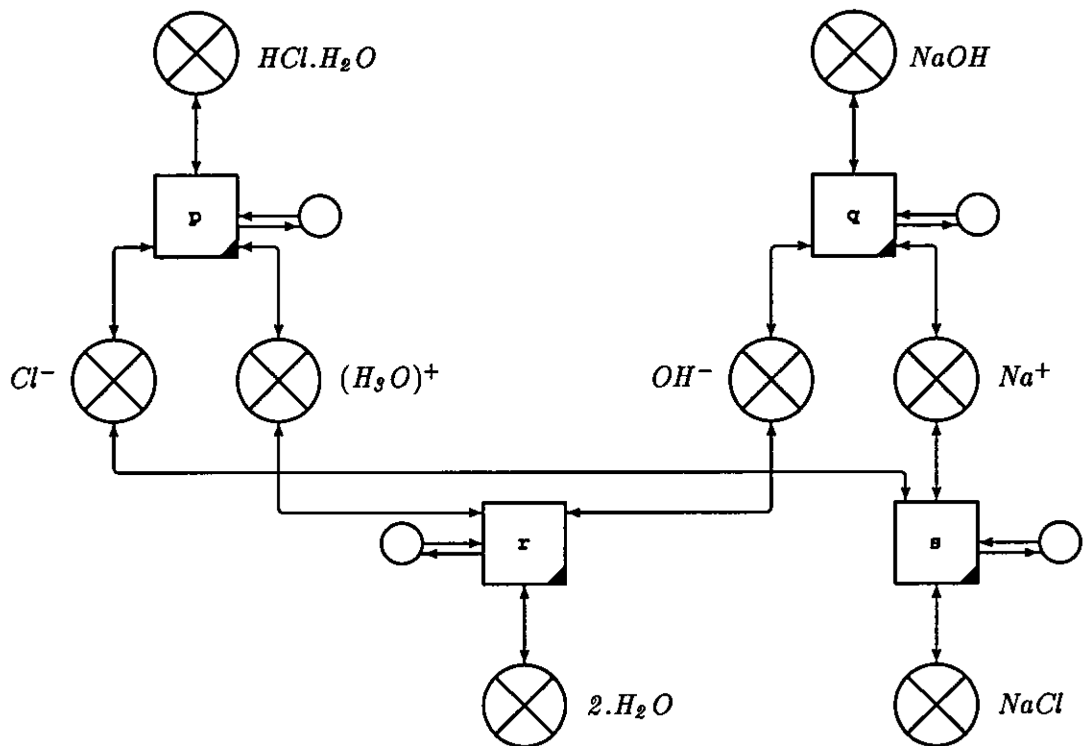

Figure 12.28: A continuous process: salt production.

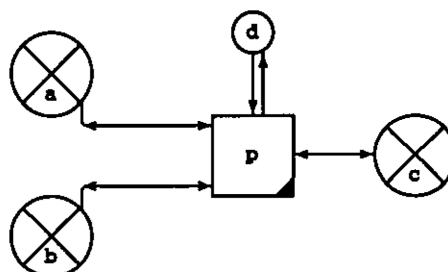

Figure 12.29: An arbitrary continuous process.

by a store, the value of the token in the store represents the amount of the chemical in that reservoir. All processors are using self-triggering and the tokens in the self-triggering places have delays that represent the time necessary to perform the chemical reactions. In figure 12.29 an arbitrary continuous process is displayed. Stores a and b contain the raw material that is transformed into the material in c. Note that we cannot see the direction of the process in the diagram because we use stores. Let us consider the processor relation of processor p. Suppose a fraction α of a unit of c is made of the material of a and a fraction $1-\alpha$ of a unit of c is made of the material of b. Further suppose that the processing time of 1 unit of c is β time units. Then the processor relation satisfies the following equations:

$$\begin{aligned} h &= min\{\frac{a}{\alpha}, \frac{b}{1-\alpha}\} \\ c' &= c + h \\ a' &= a - \alpha \times h \\ b' &= b - (1-\alpha) \times h \\ delay(d) &= \beta \times h \end{aligned}$$

Note that h is the maximal amount of c that can be produced and that we may break up the process into more steps, by not producing the maximal amount of the token in c.

13. Communication with an environment.

environment

As we have remarked before, we have to replace an environment of a (model of) system by approximations of context actors, in order to be able to simulate the behavior of the system. The approximations of the context actors should have a behavior that is at least as "rich" or "wild" as the behavior of the real environment in order to be able to test the system. There are in principal three different ways to model the context actors as displayed in the three cases of figure 12.30:

- without *feed back* as displayed in the first case
- with memoryless feed back, as in the second case
- feed back with memory, as in the last case.

In all three cases p represents the system for which we have to create an environment. The processors q and r are both total and complete. In the initial states we assume tokens in places a and s only.

In the first case processor r just consumes the output of the system and processor q produces input without any concern of the output of p. Note that q fires because of self-triggering via place s. In s may be one or more tokens, and q produces a new one in every firing. A slight variation of this solution is one in which processor q does not produce tokens for s. In that case the environment at some point in time stops producing new input. This case can also be modeled by putting all the input tokens immediately in place a with an appropriate time stamp.

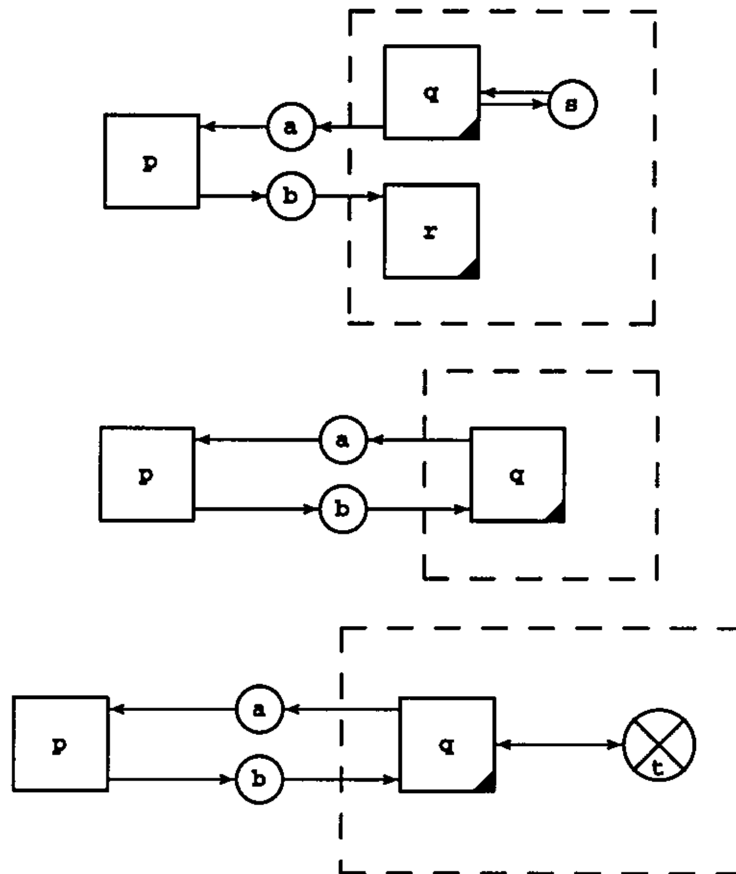

Figure 12.30: Three ways to model an environment.

In the second case we see that the environment only produces input after it has received output from the system: the communication follows a simple protocol. In this case the new input may depend on the last output but not on the history of outputs of the system.

In the last case the new input is allowed to depend on the whole history of the output of system p. Note that the last solution has a store t, which may of course be changed into a place like in the first case. However, in the first case we may not replace place s by a store since we do not want processors to be triggered by stores.

12.3 Structured networks

The actor framework describes a very large class of models. It is useful to distinguish *types* of actor models, i.e. subsets of the set of all actor models having some common properties. Such a type division may be based on:

valueless actor models

- class model: an important type is formed by the *valueless actor models*; these actor models only have one object class, which contains only one object;

- delay structure: an important type is formed by the *timeless actor models*; in these actor models the tokens in the initial state should have a time stamp equal to zero and all delays assigned by processors should be zero too;

timeless actor models

- processor characteristics: an important type is the one in which all processors are complete, total and functional;
- network structure: this subdivision is based on the structure of the graph only, there are several important types based on different graph structures;
- state structure: this subdivision is based on the maximum number of tokens per place.

We will describe several types and we will consider transformations of actor models belonging to one type into another type.

classical Petri nets

The first type we will consider is the type of *classical Petri nets* (also called *place/transition nets*). Recall that a classical Petri net is a *timeless* and *valueless* actor model, i.e. all tokens have time stamp 0 and there is only one complex class with only one complex in it. Furthermore all processors are complete and total. Note that for valueless actor models the functionality of the processor relation is not important. For this type of actor models nice and useful analysis techniques are available, as studied later, but we give already some intuitive ideas of some of these properties. However, the expressive power and comfort are very little. Nevertheless it is sometimes possible to transform actor models where tokens have values to classical Petri nets, which allows analysis of these actor models by the techniques for classical Petri nets. The intersection of the actor model types we define below and classical Petri nets are well-studied in literature. They have interesting behavioral properties.

The definition of a (flat) actor model is given in part II. Here we only need to know that L denotes the set of places, P is the set of processors, $I(p)$ is the set of input connectors of processor p, $O(p)$ is the set of output connectors of processor p and M_p is a function assigning the connectors of p to places.

Let L' denote the set of all channels in a flat actor model (so $L \backslash L'$ is the set of all stores). We exclude stores from the structural properties, because they do not influence the enabling of processors: they are always available. In order to define properties of actor models we need some definitions. For $p \in P$ the symbol $p\bullet$ is the set of output channels:

$$p\bullet = \{l \in L' \mid \exists x \in O(p) : M_p(x) = l\}$$

and $\bullet p$ is the set of input channels:

$$\bullet p = \{l \in L' \mid \exists x \in I(p) : M_p(x) = l\}.$$

For $l \in L'$ the symbol $l\bullet$ is the set of processors for which l is an input place:

$$l\bullet = \{p \in P \mid \exists x \in I(p) : M_p(x) = l\}$$

and $\bullet l$ is the set of processors for which l is an output place:

$$\bullet l = \{p \in P \mid \exists x \in O(p) : M_p(x) = l\}.$$

free choice nets

The actor models we will consider now are called *free choice nets*. We assume the processors are total and complete when we are dealing with free choice nets. A *free choice net* is an actor model such that:

$$\forall p \in P, l \in L' : \#\{x \in I(p) \mid M_p(x) = l\} \leq 1$$

and

$$\forall l \in L' : (\#(l\bullet) \leq 1 \vee \forall p \in P : p \in l\bullet \Rightarrow \bullet p = \{l\}).$$

In words, in a free choice net every processor is connected to a place with at most one connector and every place is either input place for only one processor, or it is input place for more processors, but then these processors have only this place as input place.

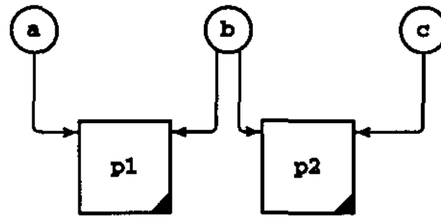

Figure 12.31: A "non-free choice" net.

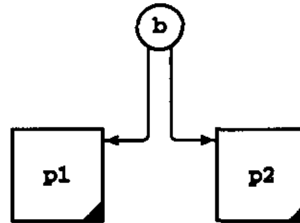

Figure 12.32: A free choice net.

In figure 12.31 an example of a "non-free choice" net is displayed. In a free choice net (that has by definition complete and total processors) the Demon is free to choose the processor that will consume a token if there is more than one possibility on base of the network. This is not the case in the example of figure 12.31, in which the token in place b cannot be consumed by processor p_2, but only by p_1 in the state where a and b have both one token and c is empty.

Next we consider three subtypes of free choice nets: conflict free nets, state machine nets and activity networks.

conflict free net

A *conflict free net* is an actor model satisfying

$$\forall l \in L' : \#(l\bullet) \leq 1.$$

In words, each place is an input place for at most one processor. It is easy to verify that a conflict free net is a free choice net.

In a conflict free net processors never have to compete for a token: there is no choice to which processor a token of a place will go. So a nice property of conflict free nets is that all processors that are enabled at some point in time, may fire at the same time, and if they do not share stores they may even fire simultaneously. Compare this property to the serializability property (cf. theorem 10.5): this theorem states that if two or more processors may fire at the same moment, they may also do this in an arbitrary order. However, for conflict free nets we do not have to determine which combinations of processors may fire simultaneously, we just have to find all processors that may fire in isolation at some moment and we know that they may fire all at that moment! An example of a conflict free net is a sequential process as studied above.

A *state machine net* is an actor model with the property

state machine net

$$\forall \in P : \#(\bullet p) = \#(p \bullet) = 1.$$

In words, each processor is connected to exactly one input and one output place. It is easy to prove that a state machine net is a free choice net.

State machine nets can be used to model *finite state machines*. Finite state machines are often used in theoretical computer science and in software engineering. In software engineering they are for instance used to specify the functionality of actors, protocols and user interfaces. Since finite state machines are a special type of actor models we may apply our framework in all cases in which finite state machines are used. A finite state machine is a state machine net with an initial state that only has one token and no stores. All processors are complete and total. Each place represents a state of the machine and each processor represents a possible transition of the machine to another state. The token indicates the state the machine is in. It is easy to see that in all states of the actor model (not to be mixed up with the state of the finite state machine) the number of tokens is one, if it starts so.

finite state machine

An example of a finite state machine is modeled in figure 12.33. Note that a state machine net can be regarded as a graph with one kind of nodes, namely places, and that the processors are considered to be arc labels. Here we see an actor model that "counts" the number of 1's in a binary sequence, i.e. it counts up to three and if it has counted three 1's it remains in its state, in case it gets a 0 it jumps to its initial state a. Channel a denotes the state of the machine after having seen a 0, b after one 1, c after two 1's and d after three or more 1's. If a 1 is received processor p fires and the machine will move to state b and if a 0 is received processor m will move the machine to a again. Similarly processors q and r move the state to places b and c respectively. Processor n moves the state to place d when a 1 is received. The other processors move the state in case a 0 is received. To understand this model we need a lot of extra information, such as the process of receiving the binary sequence, not displayed in the diagram. In our framework it is easy to extend the model with an environment modeling the extra information. In figure 12.34 the augmented model

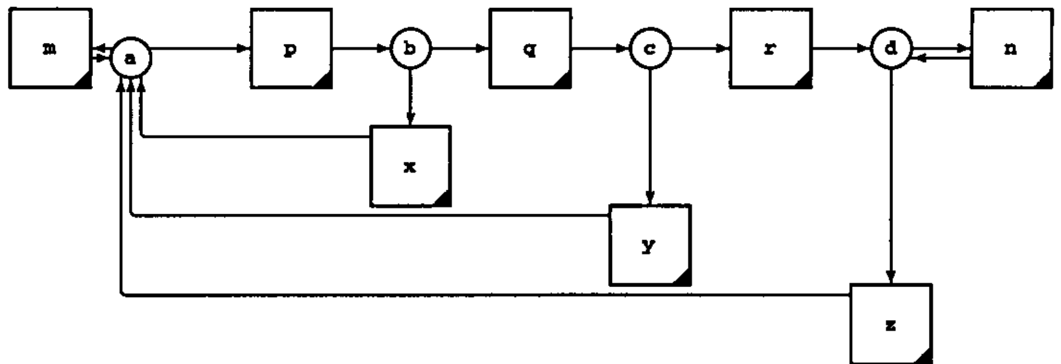

Figure 12.33: A finite state machine.

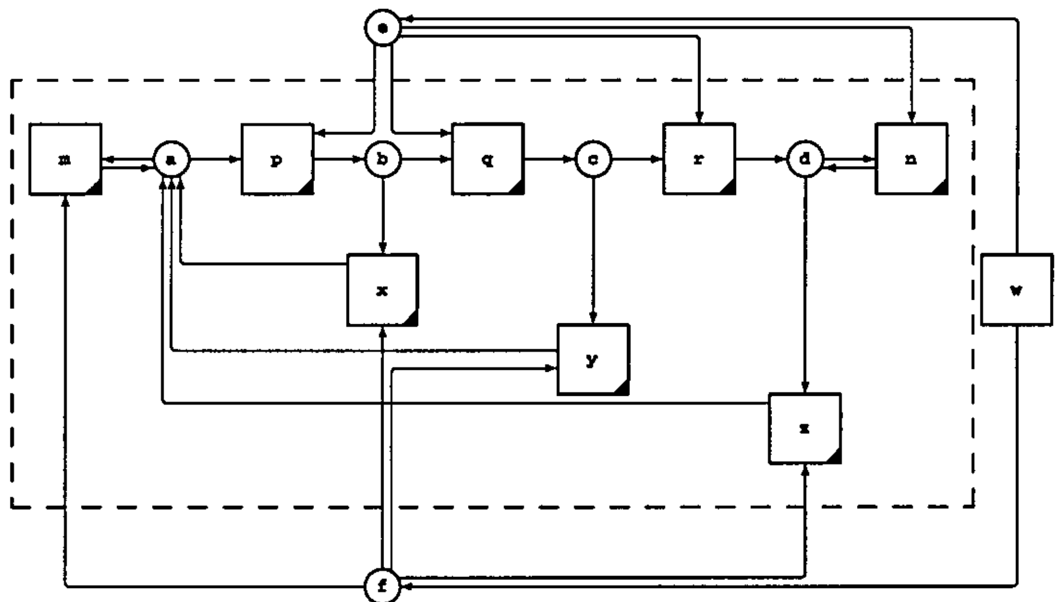

Figure 12.34: The finite state machine in an environment.

is displayed. Here places e and f get the 1's and 0's respectively from an actor w. The processors p, q , r and n consume the 1's while the other processors consume the 0's if they are enabled. We assume actor w does the appropriate selection of the elements of the binary sequence and we assume that w produces the tokens for e and f with a delay that is larger than the time the finite state machine needs to perform the transitions.

activity network

An *activity network* is an actor model with the property

$$\forall l \in L' : \#(\bullet l) = \#(l \bullet) = 1.$$

In words, each place is an input place for exactly one processor and an output place for exactly one processor. It is easy to prove that an activity network is a conflict free net.

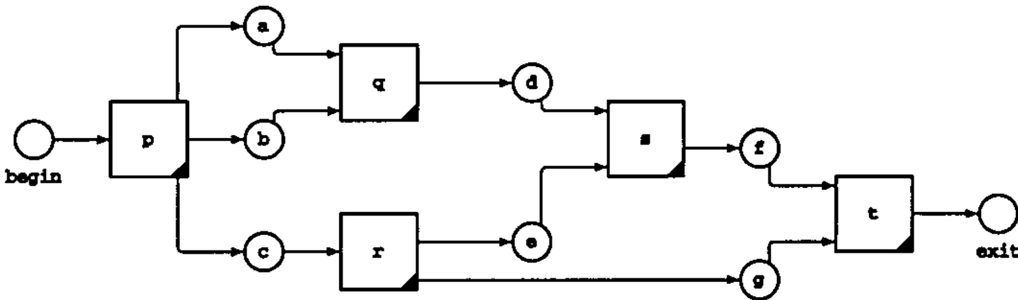

Figure 12.35: A PERT network.

An activity network is also known under the name *marked graph* or *PERT network*. (PERT is an acronym for Program Evaluation and Review Technique.) The last name is used only if the network is a-cyclic and if there are extra two places, say *begin* and *end*, such that $\bullet begin = \emptyset$ and $end\bullet = \emptyset$. (Formally a PERT network is no activity network, due to the extra places.) Each processor represents the start or the end of an activity while each place represents the execution of an activity. Activity nets can be used to model *parallel processes* with *precedence constraints*. When we give each token a delay that corresponds to the time an activity takes, the simulation of the model gives us the *earliest* possible completion time of the set of activities, also called the "project". In figure 12.35 an example is displayed. Note that an activity network can be seen as a graph with processors as nodes and arcs labeled with places. In this network the initial state is one token in *begin*. Then processor p marks the start of the activities a, b and c in parallel. Only if a and b are ready processor q can start activity d. Finally all activities are done if processor t produces a token for *end*.

Another important type of actor models, based on the state structure, is called *bounded nets*. These actor models are characterized by properties of the maximal number of tokens per place.

bounded nets

A *k-bounded net* is an actor model where each place has at most k tokens in each state, provided the initial state has this property. A *safe net* is a 1-bounded net.

k-bounded net
safe net

We have already seen example of safe nets: a state machine net is safe, because there is at most one token in the network; an activity net is also safe. If the number of values the tokens may get in a k-bounded net is finite, then the number of states if finite too and it will be possible to model the network as a finite state machine.

12.4 Net transformations

We will now consider several ways to transform an actor model into one with another structure.

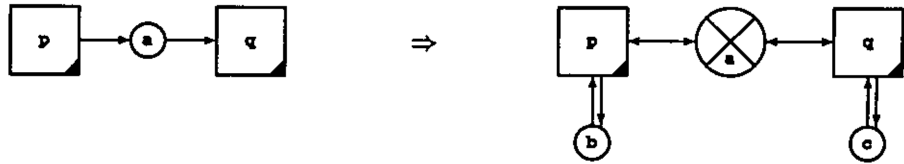

Figure 12.36: Transforming an actor model into one with only self-triggering.

Transforming actor models into models with only self-triggering.
Suppose we have an arbitrary actor model and we want to get rid of communication by means of places. A reason for this could be that it might be easier to realize a system in which processors inspect memory at their own time instead of having synchronization problems. Then we can transform such a network into a network with only one place per processor that is used exclusively by this processor, both as input and output place. The transformation proceeds along the following lines:

self-triggering only

1. Replace each place by a store.
2. Give the store a complex class such that tokens of the original place can be clustered into a complex of this class and that the complex can be decomposed into the original components. I.e. if the complex class is represented by a value type T, then the complex in the store can be represented by $\mathbb{F}(T)$ or T^*.
3. Introduce for every processor a place which contains in the initial state only one object, which is a valueless complex.
4. Modify the processors by adding a *pre-processing* phase in which they select one token from the complex in each of their stores that was an input place and a *post-processing* phase in which they pack the objects they produce into the complexes of the stores that where output places.

In case time plays a role it is necessary to represent the time stamps of the original tokens in the complexes of the stores in order to be able to pick them in the right order. The new network should be similar to the original one, see definition 8.8. The new network inspects its stores by means of self-triggering or *polling*. The tokens in the new private places may get a delay to model the polling intervals. An example is given in figure 12.36.

polling

In a similar way we can transform an actor model to have only one input place that is either a self-triggering place as above or an output place of some other processor, which means that some processors are only activated by others and some by self-triggering.

Transforming actor models into models without shared stores.
Now we consider a situation that is almost the complement of the former one: we have a network in which processors share stores and we

would like to transform this net into a net in which stores are private for a processor. A store can be accessed by only one processor in an event, so we should incorporate in the transformed network a mutual exclusion mechanism.

no shared stores

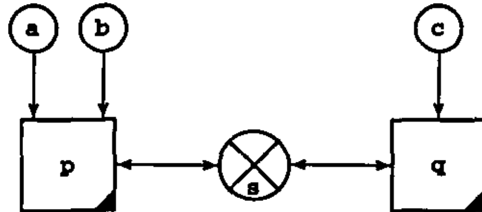

Figure 12.37: Network with shared store.

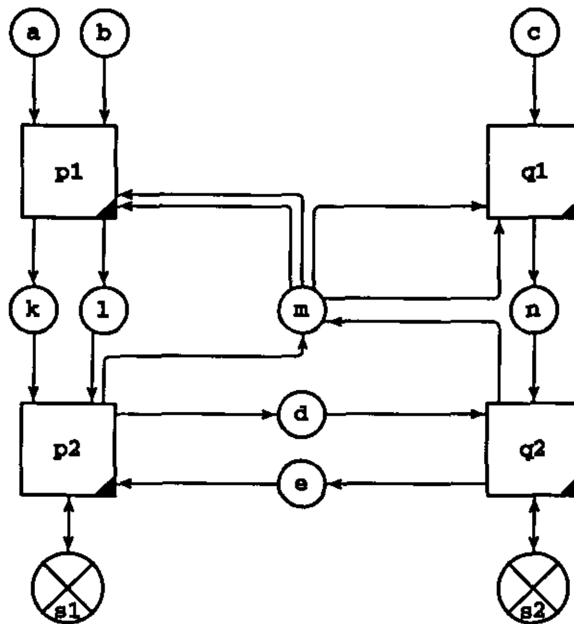

Figure 12.38: Network without shared store.

In figure 12.37 and figure 12.38 we see an example. In the network of figure 12.37 processors p and q share a store s. We assume both processors are complete. The transformed network should keep the values of the two tokens in s_1 and s_2 the same, as much as possible. In figure 12.38 we see the transformed network in which processor p is replaced by two processors: p_1 that takes care of the access control and p_2 that performs the update of the store s_1, which has the same type as s. For processor q we see a similar transformation. In the initial state place m contains two valueless tokens that are needed both by p_1 or q_1 if they want to initiate an update. After p_2 and q_2 have done the updates they return one (valueless) token to place m. Further we see two new places d and e that are needed to transfer the update of one processor to the other. (Note that p and q may perform totally different kinds

of updates). Processors p_1 and q_1 are complete but p_2 and q_2 are not input complete: p_2 is either consuming a token from place k or from e. The same counts for q_2. However they always give a token to place m. So if the two tokens of m are consumed by, for instance p_1, then one token is returned after the update of s_1 by p_2 and the other one by q_2 after the update of s_2. Processors p_1 and q_1 have a very simple processor relation: they pass the tokens they consume from a, b and c to k, l and n respectively. For processors p_2 and q_2 there are several transformations possible, depending on what the original processors p and q do. A trivial, but in many cases not practical solution is to let p_2 give the new value of the token in store s_1 to q_2 and vice versa. This modification of p_2 and q_2 is easy. In case the token in the stores is a set and the update is just the addition or deletion of an element of that set, processors p_2 and q_2 only have to exchange their updates and not the new value of the stored token.

The solution we have considered here can easily be generalized to three or more processors sharing one store. Another generalization is the case in which we have three processors that share two different stores pairwise. We can use a similar transformation here.

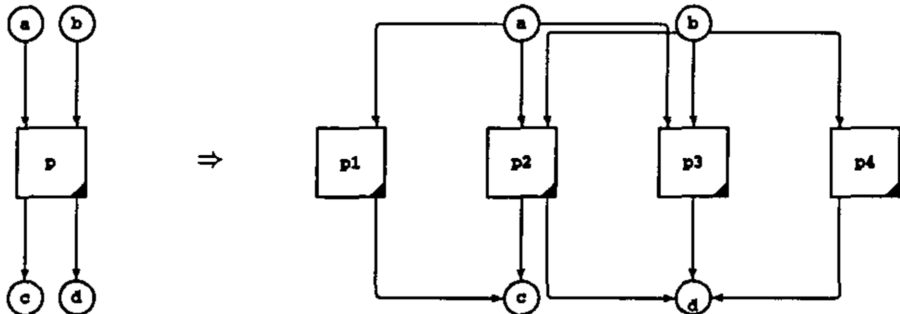

Figure 12.39: Transforming an incomplete processor.

Transforming incomplete processors into complete ones.

complete processors

Sometimes it is nice to have an actor model with complete processors, for instance to analyze behavioral properties of a classical Petri net after discarding the values of tokens. If we start with a net with incomplete processors we can transform it into one with only complete processors. Consider the processor relation R_p of a processor p. (In definition 10.3 R_p is defined as a set of firing rules; each firing rule describes a possible firing of processor p, i.e. a possible combination of consumed and produced tokens and the corresponding input and output connectors.) For all different combinations of input and output connectors involved in a transition we define a new processor.

In figure 12.39 we see on the left-hand side that p has two input and two output places. Assume its processor relation prescribes that it may fire for the following combinations of connectors: $\{a, c\}, \{a, b, c, d\}, \{a, b, d\}$ and $\{b, d\}$. This results in the net on the right-hand side in a processor

for each combination of connectors.

In general, let

$$K = \{dom(f) | f \in R_p\}$$

denote the sets of combinations of connectors for which p is able to fire, then we have to create a processor p_k for each element $k \in K$ with processor relation

$$R_{p_k} = \{f \in R_p | dom(f) = k\}.$$

Note that K is always finite, even if R_p is infinite. Processor p_k has k as its connectors. The following property is easy to verify.

Theorem 12.1 If processor p is functional then for all $k \in K$ processor p_k is functional.
□

If p is total, then p_k is not necessarily total too. Instead of decomposing a processor like we did here, we may also *cluster* processors by taking the union of their processor relations in order to reduce the number of processors! In fact we can transform each net into a net with only one processor.

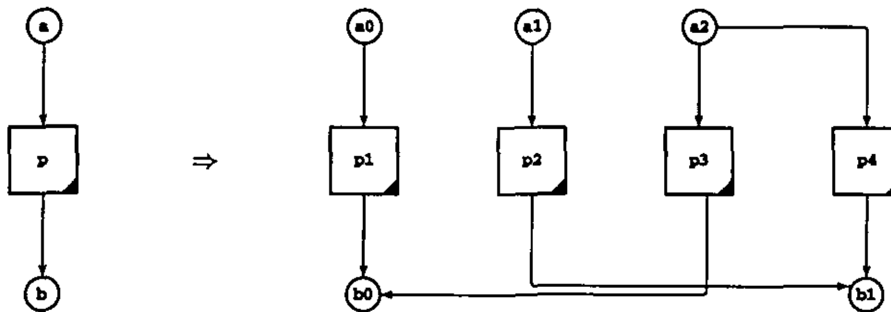

Figure 12.40: Transforming a net into a valueless net.

Theorem 12.2 n actor model with incomplete processors and the transformed complete actor model are bisilimar with respect to the identity relation over $St \times T$.

Proof. First note that both systems have the same state spaces. Every transition of one system can also be made by the other.
□

Transforming nets into valueless nets.

valueless nets

As said before, there are some useful analysis methods for classical Petri nets. If we want to apply them to nets in which tokens have values we may discard their values. However, then we loose information. In case the number of values that play a role in the transition relations is finite, we may transform the actor model into another actor model with only valueless tokens, without loss of information. The transformation is

analogous to the one above. In figure 12.40 an example is displayed: left the original net and right the transformed one. The processor relation of this net is given in the following table:

f	a	b
1	0	0
2	1	1
3	2	0
4	2	1

active domains

From this table we see that place a only gets values from $\{0, 1, 2\}$ and b from $\{0, 1\}$ (we assume that the environment gives no other values). We call these sets the *active domains* of the places, i.e. the subsets of their types that are actively used. In the transformed net each place is "copied" as many times as the size of its active domain. Further we create a new processor p_f for each function $f \in R_p$ and we connect it to the places according the domain of p. In cases where the object universe is finite we can apply this transformation.

Theorem 12.3 et an actor model A with a finite object universe OU_A be given and let B be the transformed actor model, according to the rules above. Then:

- OU_B is a singleton,
- $St_B \subset ID \to OU_B \times T \times (L \times OU_A)$,
- A and B are *bisimilar* with respect to $C = \{((s,t),(s',t')) \mid s \in St_A \;\wedge\; s' \in St_B \;\wedge\; t \in T \;\wedge\; dom(s) = dom(s') \;\wedge$

  $\forall i \in dom(s) : \pi_2(s(i)) = \pi_2(s'(i)) \wedge \pi_3(s(i)) = (\pi_3(s(i)), \pi_1(s(i)))\}$.

Proof. The state spaces are isomorphic and each transition of one system can be made by the other (to the corresponding state).
□

Chapter 13

Object Modeling

Object modeling is the activity of making an object model, including value types for simplex classes and constraints and without definition of "clever" value types for complex classes. Assignment of complex classes to places is done when the actor model is available and the value types of complex classes are determined in the specification phase; both activities are studied in the next chapter. In this chapter we consider the process of making an object model after reality, some characteristic modeling problems and methods to transform an object model from one framework into another. We start with some techniques for specifying constraints.

Constraint specification

Constraints are very important because an object model without constraints is often not sophisticated enough to express the structure of the real world state space. Constraints are either expressed graphically or in the specification language. We distinguish *local* and *global* constraints. The local constraints concern all complexes of a complex class, while the global constraints concern all states. Each state determines one universal complex in the following way. Let d and e be two arbitrary complexes of complex classes m and n respectively. Then we define a universal complex c as the *union* of d and e, hence

local constraint
global constraint

$$\forall x \in SN \cup RN : c(x) = d(x) \cup e(x)$$

where we define $d(x) = \emptyset$ if $x \notin CB(n) \cup CR(n)$ and $e(x)$ is defined similarly. (Note that the union of two complexes of the same class is itself a complex of the same class, although constraints might be forced.) So for a state s we define a universal complex c as the union of all complexes in s:

$$c = \bigcup_{i \in \mathrm{dom}(s)} \pi_1(s(i))$$

An important global constraint is that this universal complex satisfies the graphical constraints for the universal complex class. We call this the *universal constraint*. This global constraint can be treated as a local constraint, although the universal complex class does not have to be assigned to a place.

universal constraint

A local constraint for a complex class n is a predicate over that class, which means that the predicate should be evaluated in the *context* of each complex that has to be verified. To specify constraints for a complex class n in the specification language we *fix* a complex c and we use the following conventions.

- We associate with every simplex class name $a \in CB(n)$ a type T_a such that the type of the simplexes in $sim(a)$ are represented by values in T_a, and we use the symbol a for the representations of simplexes in $c(a)$. In most cases the choice for T_a is free because we do not apply any particular function to the values of T_a. In a few cases we need that T_a is a quantity (for instance represented by $\mathbb{N}$) or that it is a set of time slots (for instance represented by $\mathbb{N} \times \mathbb{N}$).

- We associate with every relationship class $r \in CR(n)$ with $DM(r) = a$ and $RG(r) = b$, a binary relation R that is obtained from $c(r)$ by replacing the elements of the pairs in $c(r)$ by their representations. Further we associate four functions with r and we call them temporary r_1, r_2, r_3 and r_4. We distinguish two cases: one where r is total and functional and one where r is not total or not functional. In the first case we can use the relationship as a single-valued function (because it always has one value). The four functions are defined by:

  - let r be functional, the r_1 is defined by:

    $$r_1(x) := apply(R, x) : T_a \Rightarrow T_b,$$

    let r does not be total and not functional, then r_1 is:

    $$r_1(x) := setapply(R, x) : T_a \Rightarrow \mathbb{F}(T_{b)},$$

  - r_2 is derived from r_1, because it is the *set-version* of r_1:
    let r be functional, then:
    $r_2(x) :=$ *if* $x = \{\}$ *then* $\{\}$ *else*

    $$ins(r_1(pick(x)), r_2(rest(x))) : \mathbb{F}(T_a) \Rightarrow \mathbb{F}(T_b)$$

    let r be not total and not functional:
    $r_2(x) :=$ *if* $x = \{\}$ *then* $\{\}$ *else*

    $$r_1(pick(x)) \cup r_2(rest(x)) : \mathbb{F}(T_a) \Rightarrow \mathbb{F}(T_b)$$

  - r_3 is defined by:

    $$r_3(y) := inverse(R, y) : T_b \Rightarrow \mathbb{F}(T_a)$$

  - r_4 is the set-version of r_3:
    $r_4(y) :=$ *if* $y = \{\}$ *then* $\{\}$ *else*

    $$r_3(pick(y)) \cup r_4(rest(y)) : \mathbb{F}(T_b) \Rightarrow \mathbb{F}(T_a)$$

Note that the functions *apply*, *setapply* and *inverse* are defined in the toolkit. We will use the overloading facility of the specification language by renaming these functions as follows: r_1 and r_2 are called r, and r_3 and r_4 are called r^{-1}.

Note that the type of an argument determines the signature of the function r or r^{-1}, so there will be no confusion. This overloading is very useful because now we do not have to distinguish between applying a function to an element or to a set. Now we may write, for example for (not functional) relationships r and q with $DM(r) = a$, $RG(r) = d$, $DM(q) = b$ and $RG(q) = d$:

$$\forall x : a \bullet \exists y : b \bullet y \in q^{-1}(r(x))$$

which means in the meta language:
$\forall c \in com(n)$:

$$\forall x \in c(a) : \exists y \in c(b) : \exists z \in c(d) : y \in D_{q,c}(z) \ \wedge \ z \in R_{r,c}(x).$$

Note that this constraint should hold for all the complexes in the complex class n.

The functions for relationships can be derived (also by a tool) from the class model. Therefore we assume they exist as soon as we have defined the class model.

The types we use for the simplex classes is not important here. In a few cases we will assume that a simplex class is a set of time slots, in which case we assume the type $\mathbb{Q} \times \mathbb{Q}$. Each pair of rationals will be interpreted as an interval. Sometimes we assume the simplex class is a set of amounts, in which case the type will be $\mathbb{Q}$. Further we only use set-theoretical functions in constraints (such as $\subset$ and $\in$).

The relationship constraints, the inheritance constraints and the tree constraints can be expressed in the specification language for each specific object model, however these predicates may be quite complex and they have a standard structure. In principle it is possible to generate these predicates automatically.

There are more frequently occurring constraints and they will be discussed in the next section as "characteristic modeling problems". We will not specify global constraints that cannot be treated as a local constraint. (If we want to do so, we have to define a type for a state space.)

13.1 Making an object model after reality

We will start with the description of two examples. They are related to the order processing and the railway station examples of chapter 12. These two descriptions determine the complex classes of the state spaces of two different systems.

Example: Factory
Consider a factory that only produces products if there is an order for it. Each product requires several construction tasks and in each task one or more components have to be assembled. The tasks for a product have a partial ordering. Each task requires some resources for some time, called the duration, to perform the task. Examples of resources are machines, vehicles and human beings. It is assumed that these resources have to be available for the whole duration of the task. There are several resources that may perform the same function. So in fact a task is specified by some functions instead of resources. A resource can be used for one task at a time, so there is no resource sharing. Components are bought from suppliers. Several suppliers may sell the same component for their own price. The factory keeps components in stock. The production schedule is just a set of operations. An operation is the execution of a task for some particular order with some particular set of resources in a particular time slot. There are two kinds of orders: customer orders and supply orders. Each order has a delivery date. An order may concern several items of several products. For components we distinguish the total number of items in stock at some day, and the number of free items, i.e. the number of items that is not assigned to an operation yet. Components of the same kind are not distinguishable, only their number counts.

Example: Railway system
Consider a railway network where track segments are defined between nodes. A node is a crossing, a switch or a semaphore. (Track segments are simply called tracks.) They are directed, i.e. trains can only use tracks in one direction. A switch connects three tracks: one fixed track, one straight track and one branching track. The fixed track is always part of the route, and from the others only one. The straight track and the fixed track form a straight line in the neighborhood of the switch, while the branching track and the fixed track form a curve. (cf. figure 13.5.) Further there are trains. A train is a temporary "cluster" of a locomotive and a sequence of wagons. A switch has at each moment one of the two positions: "straight" or "branching", and a semaphore has one of three status values: "green", "orange" or "red". A train also has a position at each moment: the track where it resides, and only one train is allowed per track.

Stepwise development
The development of an object model proceeds upon the following steps:

Step 1: Determine relevant *entities*.
All entities, that can be named by a *noun* are either objects or actors. If they are objects, they can be either simplexes or complexes. In fact the nouns indicate *classes*. So the noun "horse" refers to the class (or set) of all horses. In a sentence we use it as "Runner is a horse" or "the horse that win the race". In the last case the sub-sentence determines

a unique element in the set of all horses. Entities have an identity (like the name "Runner" for a horse). A way to find the relevant classes is to collect all nouns that appear in documentation over the system to be modeled, i.e. in forms, instructions, reports etc. This is a *syntactical analysis* of written or spoken text. Of course, if the systems engineer has already some knowledge of the type of systems to which the system he has to model belongs, he probably knows most of the relevant nouns already. The first step ends with the exclusion of actors. An entity is regarded as an actor in one of the following cases:

- it is in the system during the whole life of the system, so the set of actors of a certain class is fixed,
- it is active, i.e. if it consumes and produces other entities (in fact objects),
- it is an event, i.e. it occurs at some point in time and it does not "live" for a time interval,
- it does not have relationships with other entities, that may change over time.

All other entities are objects. Remember from chapter 12 that processors that represent some physical entity can be split into a processor that represents the *activity* of the entity and a place in which the token represents the physical entity itself. (The token is consumed and (re-)produced in every execution of the processor.) So the all "things" can be modeled as objects, in which case all processors represent activities. We distinguish *compound* or *molecular* objects, called *complexes* and *atomic* objects, called *simplexes*. Actors are further studied in actor modeling.

compound object
molecular object

Step 2: Determine the *simplex classes*.
We distinguish *concrete* simplexes, like the resources and the locomotives in the examples above and *abstract* simplexes, like a task, an operation, a time slot or a train position. A third category of simplexes are *information* simplexes. They refer to either concrete or abstract entities. Concrete simplexes are physical entities, while abstract simplexes are activities, events, qualifications, quantities, agreements, instructions or concepts.

concrete simplex
abstract simplex
information simplex

To decide which objects are simplexes we give the following rules:

- simplexes have a unique value that is independent of other objects and that can be used to identify them,
- simplexes are *atomic*, i.e. we can not "look inside" them to discover other objects, so we do not allow functions that produce values of other simplexes if applied to the value of a simplex,
- in different states of the system the set of simplexes of the same class may be different, may contain more than one simplex and must be finite.

Even if an object consists of parts that are considered as well as objects, it can be considered as a simplex if it can be given an independent, atomic value ("train" is an example of this). In that case we model its components as simplexes as well and we use relationships to express that one is part of another.
We may classify simplex classes into three groups.

attribute simplex class

- *Attribute* simplex classes.
  A simplex class n is an attribute simplex class if n is not a domain class of any relationship class, i.e. $\forall r \in RN : n \neq DM(r)$. Attribute simplex classes do not play an important role because their simplexes have no properties of their own. Therefore we neglect them sometimes in the first stages of development of an object model.

association simplex class

- *Association* simplex classes.
  A simplex class n is an association simplex class, if all relationship classes with n as domain class, together form a minimal key. Further it is required that n is not the range simplex class of some relationship. Simplexes of these classes have only one role: the coupling of other simplexes. They usually are found in a later stage of the development process.

entity simplex class

- *Entity* simplex class.
  A simplex class n is an entity simplex class if it is not an attribute simplex class or an association simplex class. Simplexes of these classes are the important simplexes, they represent the entities we see in the real world. The attribute simplex classes are often defined in a late stage. (It is useful to distinguish these different types of simplex classes by different graphical symbols; for attributes often circles are chosen and for associations diamonds.)

This classification is useful in the design process: start with the entity simplex classes.

Step 3: Determine *relationship classes.*
Relationships connect simplexes. If a simplex is connected to another simplex we consider this as a *property* of these simplexes. Relationships are labeled with a *verb*, often in two forms: active and passive (cf chapter 4). Relationships express a *status quo*, for instance "is made of", "belongs to" or "located at". Relationships belong to classes, like simplexes and all relationships of one class connect only simplexes of one simplex class to simplexes of one (not necessarily different) other simplex. Relationships have a *direction*, we choose the verb such that the sentence made of the noun at the domain of the relationship, followed by the verb and the noun at the range of the relationship, form a sentence in active form. There is some freedom in the choice of the direction of a relationship class. (Remember that the inverse of a functional relationship is an injective relationship.) The choice is based on

the intuitive meaning of the relationship: it is a property of the domain class of the relationship class. Further the choice may be influenced by the use of the relationship in key, exclusion and tree constraints.

Step 4: Determine *complex classes.*
A complex is a cluster of simplexes and relationships. The relationships should connect only simplexes that belong to the complex as well. Complexes are usually defined if an actor model is already available, because the complex classes partition the simplex and relationship classes over the places.

Often a set of simplexes forms a complex, for example a set of trains can be a complex in the railway system. Many complex classes satisfy a *tree constraint*, which means that complexes of such a class have one *root* simplex that identifies the complex and that gives "access" to all other simplexes in the complex. In the railway system for instance, there is a complex class called "train cluster" the complexes of which have a train simplex as root and a locomotive and a set (in fact a sequence) of wagons in the *body*. So here we distinguish the locomotive connected to the wagons as different from the train itself, which implies that there might be different trains with the same locomotive and the same set of wagons. (Of course these different trains will not exist at the same time, but that is not relevant here). It was also possible to use the noun "train" only for the cluster consisting of wagons with a locomotive as root, however then two trains are identical if they have the same locomotive and the same set of wagons. So we have the freedom to introduce a simplex class to identify a cluster of simplexes connected by relationships. We distinguish concrete, abstract and information complexes as well as combinations.

All simplex classes and all relationship classes should occur in at least one complex class, since only complex classes are considered in a state. So if for instance a relationship class does not appear in a complex class, then either the relationship class is irrelevant or the complex classes are not defined completely. This requirement provides a check point for the systems engineer.

Step 5: Determine *value types* for simplex classes.
This activity may occur immediately after the definition of simplex classes, however we only need these value types sometimes in constraints. Note that the value types for simplex classes define the function *sim* of the instance model of an object model (cf. 9.2).

Many constraints do not refer to the values of simplexes at all, but sometimes we need some *properties* of the values of the simplexes. For instance if the simplex class denotes quantities or time slots we may need a value type to express constraints because we have to apply functions to the simplexes. The values of the simplexes are the only things we know of the simplexes, so the simplexes are identified by there values. If we do not give value types for simplex classes in examples, they are irrelevant for constraints. We avoid the use of simplex values in constraints as

much as possible for the same reasons why we avoid parameters hard-coded in programs. Note that complex classes are completely defined if the value types of the simplex classes are known. In fact we have a "default" value type for them. However in specifications we might want to use another, more sophisticated representation for complexes. (This will be clarified in the next chapter.) In object modeling we need no other representations of complex classes.

Step 6: Determine *constraints*.
First we start the *relationship constraints*, which have a graphical representation. They are supposed to hold for all relevant complex classes, i.e. complex classes that contain the relationships involved. If we discover a key constraint we always look for a *minimal key constraint*, i.e. if n is a simplex class then $a \in DK(n)$ is a minimal domain key if

minimal key constraint

$$\forall b \in DK(n) : b \subset a \Rightarrow a = b$$

For range keys the definition is similar. In the same way we search for *maximal exclusion constraints*, i.e. for exclusion constraints that are not contained in larger exclusion constraints.

maximal exclusion constraints

Next we determine *inheritance constraints*. We use inheritance to distinguish different subclasses of a simplex class in case simplexes of a sub class have special relationships the first simplex class does not have. For complex classes we determine the *tree constraints* as discussed above.

Next we search for constraints that can not be expressed as one of the above mentioned constraints; for these we use predicates in the specification language. We mention four ways to find (some of) them.

- Search for *cycles*.
  If there is a cycle in the class model, then we can start with a simplex in one of the simplex classes and we can follow two different routes to a simplex or a set of simplexes in another simplex class (in the cycle). Are these simplexes the same or should these (sets of) simplexes be disjoint? One cycle gives rise to many of these questions. Here the direction of the relationships is irrelevant.

- Search for *time orders*.
  If "time" or "time slot" is a simplex class, then often the simplex classes related with them have constraints with respect to time. For instance if two operations have a same time slot then the resources connected to them should not be the same, because a resource can be involved in only one operation at the same time.

- Search for *balances*.
  If quantities are simplex classes then there is often some balance required. For instance if there is a simplex class "order", which is related to a simplex class "quantity" and if there is a simplex class "order item" which is also related to "quantity", then there might

be a constraint that requires that the quantity of an order is the sum of the quantities of the related order items. (Note that an order consists of order items.) Note that such constraints can be avoided if there was no relationship between order and quantity. This relationship is indeed superfluous because the sum can be computed. However if not all order items have a quantity (yet) we have to keep both relationships with "quantity".

- Search for *temporal inconsistency*.
  Often we consider a complex in several stages of development. For instance an order is already defined but not all of its order items. In such cases we have to *drop* constraints because we allow also complexes that will be correct after some modifications. (We use inheritance to solve this problem (cf. section 13.2.)

  Finally we look for global constraints, such as the *universal constraints* and constraints which require that simplexes representing physical entities are unique in a state (i.e. a physical entity cannot be in two different places in the same state).

We will illustrate this development process for the two examples described above.

Example: Factory (continued)
We start with listing the relevant nouns of the description: product, order, task, component, resource, function, supplier, price, stock, schedule, operation, time slot, duration and delivery date. Note that "factory" is not a relevant noun because there is only one factory that stays the same during the whole life time of the system. The factory may be considered as the top-level actor. We decide that a "schedule" is just a set of "operations" so we will not define a simplex class for it. In figure 13.1 we see all the other nouns as simplex classes. Two simplex classes require some elucidation: "customer order item" and "supply order item". At first sight they seem to be superfluous because we have already "customer order" and "supply order". However an order is in fact a complex object and it contains for each product or component a "sub-order", which is an object itself. Further we see a number of relationship classes labeled with characters. It is not difficult to find suitable verbs for each relationship class, for instance: a gets "intended for", b gets "belongs to", d gets "concerns" and e gets "executed for". Only relationship k requires some clarification; it is called "predecessors" and it assigns to a task the set of all tasks that are immediate predecessors of the task. With relationships we should be as *thrifty* as possible; we could for instance define a relationship between "product" and "component" that denotes the components from which the product is made off. However that relationship is *derivable* from h and l because h is giving all the tasks to be performed for the product and l gives all the components needed for these tasks, so it is *redundant*. There is a lot of freedom in the choice of relationship classes. However

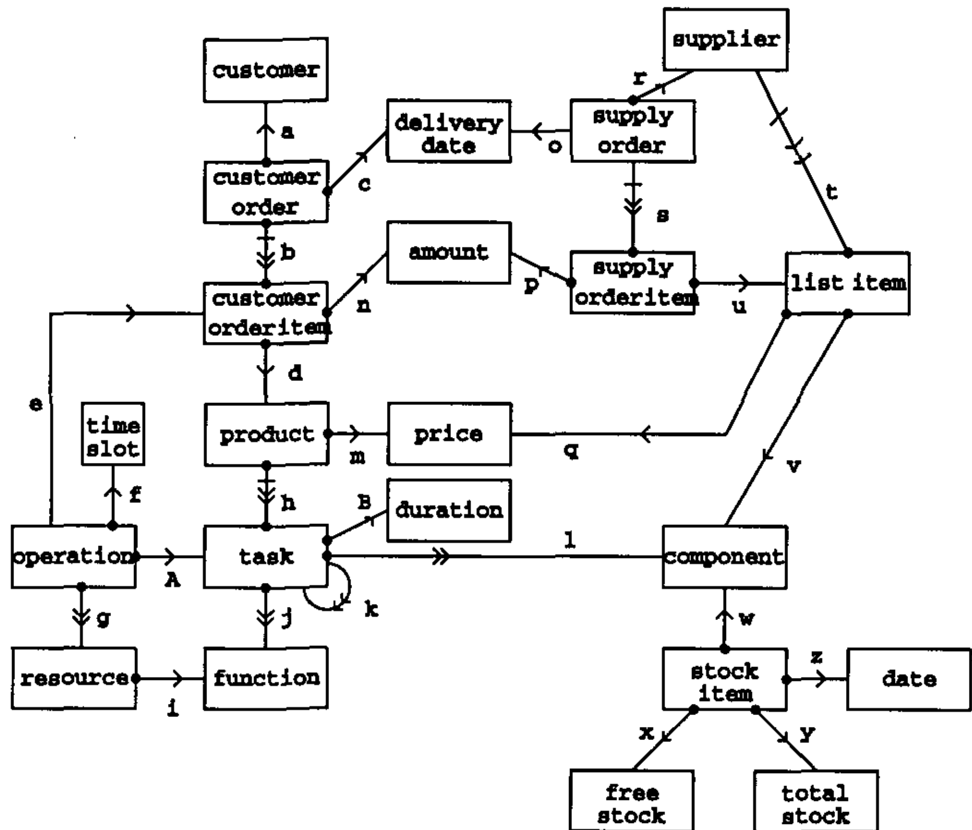

Figure 13.1: Simplex diagram for the factory.

if we introduce redundant relationships we also introduce constraints! For instance if we would have introduced a relationship class hl from "product" to "component" then we should have defined the constraint

$$\forall p : product \bullet l(h(p)) = hl(p)$$

Instead of introducing the relationship hl and this constraint, we may consider this constraint as the *specification* of the function hl and we can use this function in other constraints or in processor specifications. So we do not need the relationship hl in the object model.

The next step is the determination of complex classes. The may be indicated in the simplex diagram, however the diagram becomes very crowded. Therefore we list them in the table of figure figure 13.2. The choices of the complexes may dependent on the actor model, in fact on the "processing". So in general it is not possible to fix the complex classes in the data modeling phase if the actor model is not ready. However some useful complex classes are listed in the following below. These complex classes have straightforward interpretations. A "BillOfMaterial" complex for instance, gives for a specific product all the tasks needed to construct the product and also the required functions and components per step. It looks like we do not need the simplex

classes "customer order" and "supply order" any more, because we have complex classes that contain all relevant information. This is not true because an order has a unique identity that is a property of all the order items in the order. This unique identity is given by the "order" simplex classes. Now it is for instance possible to have two different "CustomerOrder" complexes with the same "delivery date", the same "amount" the same "customer order item"'s etc.

It is easy to verify that all relationship and simplex classes are covered by complexes. The value types for simplex classes are easy to define. Simplex classes with names like "amount", "price", "free stock" and "total stock" get the rationals or integers as type. Simplex class "date" could get a product $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ as value type, denoting the day, the month and the year respectively. Similarly "time slot" gets the value type $\mathbb{Q} \times \mathbb{Q}$ denoting the interval bounds of the time slot.

The final step is to determine constraints. The relationship constraints speak for themselves. We did not require that relationships are surjective. This may be added. At least for the complex class "Schedule" this would avoid for instance dangling "tasks" in a "Schedule" complex. There are no key, exclusion or inheritance constraints here and the tree constraints are already indicated in the table above. So we are only looking for additional constraints. We start looking for *cycles*. Consider for instance the cycle formed by the relationships $\langle \alpha, j, i, g \rangle$. If we follow two paths from "operation" to "function", one via $\langle g, i \rangle$ and one via $\langle \alpha, j \rangle$ then we should get the same set of functions in both cases. The specification of this constraint is

$$\forall x : \mathit{operation} \bullet i(g(x)) = j(\alpha(x))$$

This constraint is defined for the *universal complex class*. Another cycle is $\langle e, d, h, \alpha \rangle$. This cycle gives:

$$\forall x : \mathit{operation} \bullet \alpha(x) \in h(d(e(x)))$$

Yet another cycle is caused by relationship k. Indeed we find two new constraints here. The first constraint says that for each task of a product all predecessor tasks should be tasks of the same product.
$\forall x : \mathit{product} \bullet \forall y : \mathit{task} \bullet$

$$y \in h(x) \Rightarrow k(y) \subset h(x)$$

The second one is more complicated. It concerns the *transitive closure* of the relationship k: a task can never be preceded by itself.
$\mathit{trans}(x) := \mathit{if}\ x = \{\}\ \mathit{then}\ x\ \mathit{else}$

$$x \cup \mathit{trans}(k(x))\ \mathit{fi} : \mathbb{F}(\mathit{task}) \Rightarrow \mathbb{F}(\mathit{task})$$

This function computes the transitive closure of the relationship k. The correctness of this definition follows from the fact that iterated application of the right-hand side of the equation (cf. chapter 24) gives a non-decreasing sequence x, $x \cup k(x)$, $x \cup k(x) \cup k^2(x) \ldots$ and there is

Complex class	Simplex classes	Relationship classes	Root class
CustomerOrder	*customerorder*		*
	customer	a	
	customerorderitem	b	
	deliverydate	c	
	amount	n	
	product	d	
Schedule	*operation*		
	timeslot	f	
	resource	g	
	task	A	
	customerorderitem	e	
BillOfMaterial	*product*		*
	task	h, k	
	component	l	
	function	j	
	price	m	
	duration	B	
SupplyOrder	*supplyorder*		*
	supplyorderitem	s	
	supplier	r	
	listitem	u	
	deliverydate	o	
	amount	p	
	price	q	
	component	v	
PriceList	*supplier*		*
	listitem	t	
	component	v	
	price	q	
StockItem	*stockitem*		*
	date	z	
	component	w	
	freestock	x	
	totalstock	y	
Resource	*resource*		*
	function	i	

Figure 13.2: Complex classes for the factory

an n such that $k^n(x) = \emptyset$ (note that we use superscript n to denote the n-th iteration).
The constraint becomes:

$$\forall x : task \bullet x \notin trans(k(x))$$

This kind of constraints occurs frequently if there is a cyclic relationship path. Not all cycles give constraints, for instance the cycle $\langle c, o, s, p, n, b \rangle$ does not give a constraint. As a last constraint we compare the duration of a task and the time slot of an operation:

$$\forall x : operation \bullet \beta(\alpha(x)) = \pi_2(f(x)) - \pi_1(f(x))$$

Note that we used *pick* here because we work with set-valued functions although we sometimes know that their value is a singleton. We are never sure that we have formulated all constraints, because we are in fact defining the laws states or complexes have to fulfill.

Example: Railway system (continued)
Again we start with listing all relevant nouns in the description. So we find: (railway) network, track (segments), node, crossing, switch, semaphore, train, locomotive, (sequence of) wagon(s), switch position, semaphore status, route and train position. We decide to consider "rail-

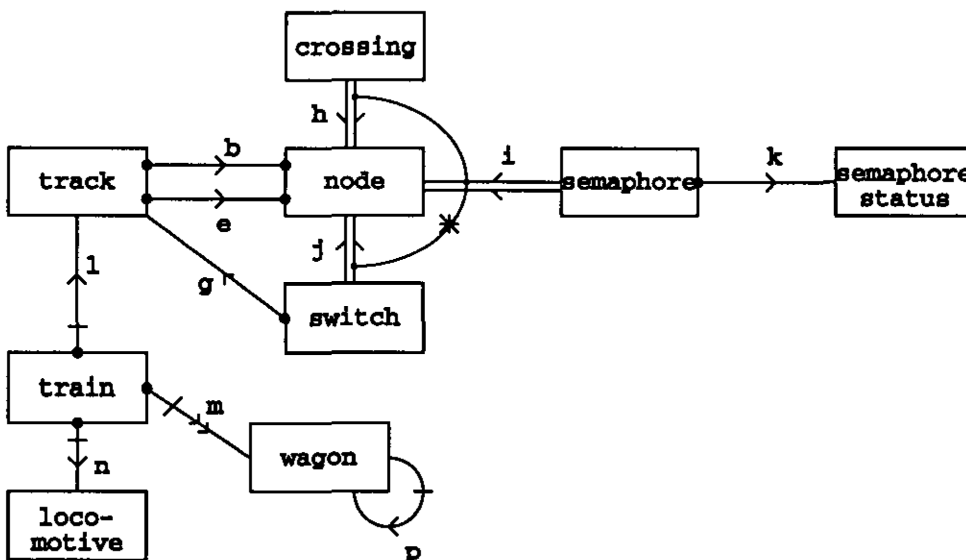

Figure 13.3: Simplex diagram for the railway system.

way network" as a set of tracks and nodes and we will not introduce a simplex class for it. The noun "route" is only used to explain the function of a switch and is considered to be irrelevant. All other nouns appear in the simplex diagram of figure 13.3, except for "switch position" and "train position". It turns out that it is possible to express the positions of trains and switches by means of relationships! However it is

Complex class	Simplex classes	Relationship classes	Root class
TrainCluster	*train*		*
	wagon	m, p	
	locomotive	n	
Network	*track*	b	
	node	e	
	crossing	h	
	semaphore	i	
	switch	j	
SwitchPos	*switch*		*
	node	j, g	
SemaphorePos	*semaphore*		*
	node	i	
	semaphorestatus	k	
TrainPosition	*train*		*
	track	l	

Figure 13.4: Complex classes for the railway system.

still possible to introduce simplex classes for them (as will be seen later) but it makes the model unnecessary complicated. In this object model we assume that in different states the sets of tracks and nodes may be different. We have chosen to represent in this model only the actual state of the railway system and not the history or the future. (Later we will see how to transform this model to express also the history and future). The relationship classes require some clarification in this case. On a crossing four track segments come together and they are part of two tracks. Each track (segment) has a direction and b indicates the begin node of a track while e denotes the end node of the segment. Relationship class l denotes the position of the train and a suitable verb for it is " is at". Similarly the switch position is expressed by g. Relationship class p determines the predecessor of each wagon, only the first wagon does not have a predecessor (or in fact the locomotive is its predecessor). The next step is the definition of the complex classes. In figure figure 13.4 the table of complex classes is given.

These complex classes have a straightforward interpretation. The value types can be chosen arbitrary except for "semaphore status" which will get a basic type with values "green" and "red". The final step is the determination of the constraints. All "standard" constraints are already given in the diagram or the complex table. Note that no two trains may reside on the same track due to the injectivity of relationship l. The injectivity of m and n implies that no two trains share locomotives or wagons. The tree constraints are already given by the table. Next we consider additional constraints. First we look for cycles. One cycle is caused by p. This gives the constraint that if a wagon is a predecessor

of another wagon then they have to belong to the same train:

$\forall t : train \bullet \forall v : wagon \bullet$

$$v \in m(t) \Rightarrow p(v) \in m(t)$$

The next constraint due to p is that there is only one wagon per train without a predecessor:

$$\forall t : train \bullet size(\{w : m(t) \mid p(w) = \{\}\}) = 1$$

Finally we observe that p should give an ordering so no wagon is preceding itself, i.e. the transitive closure of p does not encloses the identity function. This can be expressed in the same way as for the "task" precedence in the example above.

The next set of cycles we observe are formed by the simplex classes "track", "node" and "switch". A node is connecting two tracks if it is a semaphore, three tracks if it is a switch and four tracks if it is a crossing:

$$\forall x : semaphore \bullet size(b^{-1}(i(x))) = 1 \ \wedge \ size(e^{-1}(i(x))) = 1$$

$$\forall x : switch \bullet (size(b^{-1}(j(x))) = 2 \ \wedge \ size(e^{-1}(j(x))) = 1)$$
$$\vee$$
$$(size(b^{-1}(j(x))) = 1 \ \wedge \ size(e^{-1}(j(x))) = 2)$$

$$\forall x : crossing \bullet size(b^{-1}(h(x))) = 2 \ \wedge \ size(e^{-1}(h(x))) = 2$$

For a switch there are two different situations displayed in figure 13.5. In the case on the left the switch is begin point of two tracks and end

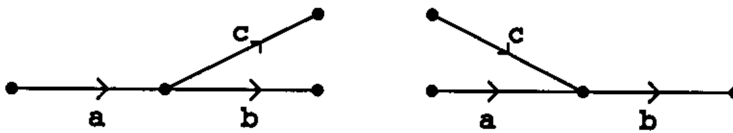

Figure 13.5: Two different switches.

point of one and in the right case the opposite. This is expressed above. We also obtain a constraint for g: it is always pointing to one of the two tracks that are either ending (right case) or beginning at the switch (left case). This is expressed by:

$$\forall x : \ switch \bullet j(x) = e(g(x)) \vee j(x) = b(g(x))$$

$$\forall x : switch \bullet e(g(x)) = j(x) \Rightarrow \exists t : track \bullet t \neq g(x) \ \wedge \ e(t) = j(x)$$

$$\forall x : \ switch \bullet f(g(x)) = j(x) \Rightarrow \exists t : track \bullet t \neq g(x) \ \wedge \ b(t) = j(x)$$

Finally we have to require that are no loops:

$$\forall t : track \bullet b(t) \neq e(t)$$

Note that we have excluded that a track comes to a dead end.

These two examples show how to perform the steps given before. They are representative for the kind of problems that appear in making an object model.

13.2 Characteristic modeling problems

Next we will consider several problems that occur in many modeling situations. Some of them appeared already in the examples above.

1. Relationships with properties
Often we have defined a relationship r between simplex classes a and b and afterwards we discover that it is not sufficient to express that there is to each simplex of a an associated set of simplexes of b, but we have to indicate also some *property* of the associated simplexes. An example of this situation can be found in the factory example. A simplified case is displayed in figure 13.6. At first sight an "order" is associated to a set of products. Later we see that each associated product has its own amount of items. A solution is to introduce a new simplex class to replace the relationship class and connect this simplex class to both original simplex classes. In figure 13.6 we introduced the simplex class "order item". This simplex class can be related to the simplex class "amount" which was not possible with the relationship b in the left-hand picture. Often we introduce immediately a key constraint, in the example a and b could form a key constraint.

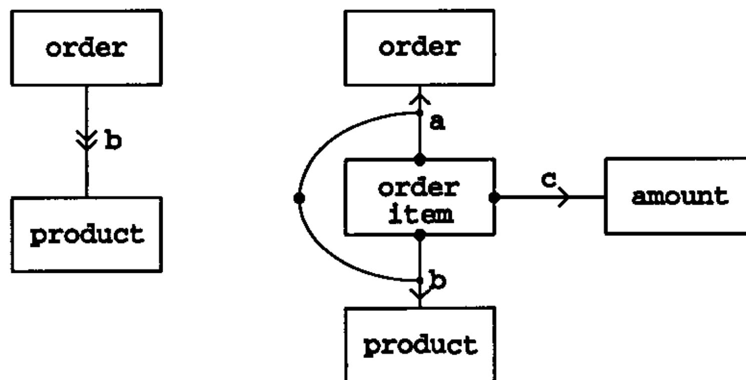

Figure 13.6: A relationship with properties.

2. Items and kinds
Sometimes we have to distinguish simplexes that represent a *kind* (or type) of objects in stead of the instances (called *items* here) themselves. In fact in the factory example the object class "product" represents

kinds of products and not the product items. Examples of product kinds are chair, car, bicycle or brand names. Examples of items of these kinds are specific chairs, cars and bicycles identified by their unique value (for instance there serial number). In figure 13.7 the general structure is displayed. The "item" simplex class always refers to the "kind" class with a total and functional relationship. A way to discover the difference

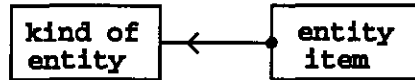

Figure 13.7: Items and kinds.

between items and kinds is to consider the value types of the simplex classes: if it is a noun, a brand name or a trade mark it is probably a kind, and if it is a proper name or a serial number it is probably an item.

3. Graphs and recursive structures

In the railway system example we modeled already a geographical network (although we did not mentioned the coordinates of the nodes). In many examples we encounter *recursive* structures such as the tree of tasks in the factory example. Here we will study these structures in isolation. In figure 13.8 we see a general graph structure and two specializations for *trees* and *sequences*. The graph is the general case

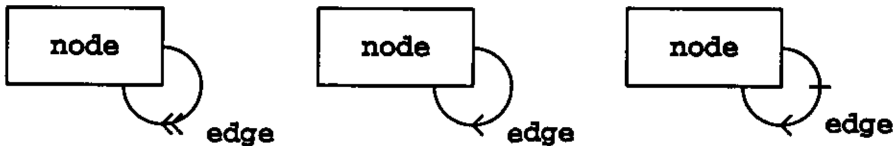

Figure 13.8: Graphs, trees and sequences.

(left most figure). Here no relationship constraints apply. The "edge" relationship determines the predecessors of a node. This choice is arbitrary; we could have decided that "edge" determines the successors of a node. The only constraint we could add is that each node is connected to at least one other (not necessarily different) node:

$$\forall x : node \bullet \exists y : node \bullet x \in edge(y) \vee y \in edge(x)$$

A subtype of the graphs is the set of *acyclic* graphs. To constrain the set of instances to acyclic graphs we introduce auxiliary functions as we have seen in the factory example.

$Trans(x) := if\ edge(x) = \{\}\ then\ x\ else$

$$x \cup Trans(edge(x))\ fi : \mathbb{F}(node) \Rightarrow \mathbb{F}(node)$$

The constraint to acyclic graphs is:

$$\forall n : node \bullet n \notin Trans(edge(n))$$

Trees are acyclic graphs with the property that the relationship "edge" is functional and assigns to every node a unique parent node. Further there is one root: the common ancestor. This is expressed by means of auxiliary function f.

$$f(x) := \textit{if } edge(x) = \{\} \textit{ then } x \textit{ else} f(pick(edge(x))) : node \Rightarrow node$$

So for $x \in node$, $f(x)$ denotes its oldest ancestor. Note that this function is correctly defined since we know that the graph is acyclic and therefore repeated application of *edge* results in the empty set. Also note that we used *pick* because *edge* is not total, so the function value might be the empty set. The constraint becomes:

$$\forall x : node \bullet \forall y : node \bullet f(x) = f(y)$$

The specialization of trees to *sequences* is simply obtained by adding the injectivity constraint to *edge* and the constraint that only one simplex does not have a predecessor (we have seen this constraint before). When the edges of graphs have properties on their own we have to introduce simplex classes as demonstrated above. (This was in fact the case in the railway system example, where "track" was the class of edges). Then the constraints have to be reformulated for the particular cases.

Trees play important roles in many practical cases. A bill of material is for instance a tree structure. Often a message in electronic data interchange is a tree. Since tree structures occur frequently we have introduced (in part II) the standard *tree constraint* for complex classes. With this constraint we can define a tree structure without additional constraints. To see how we could use this standard constraint, consider the example of figure 13.9. Here the simplex class "root" forms the

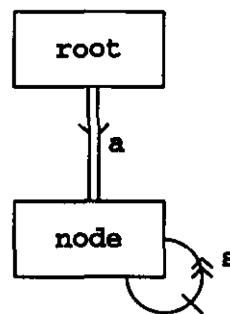

Figure 13.9: Tree constraint.

root simplex class. Further s determines all successors of a node . The injectivity constraint for s and the constraint that the root is not a successor of any node:

$$\forall x : root \bullet \forall y : node \bullet a(x) \notin s(y)$$

guarantee that there are no cycles, not even if we discard the direction of the edges. The fact that there is one root (of class "root") and the

fact that each simplex is reachable from this root constrains the complex class to trees. (The proof of this statement is an exercise.)

We consider an example of a *recursive structure* involving trees. In this example a complex class represents a syntax definition for arithmetical terms:

$$term ::= constant \mid variable \mid \underline{(}term\ binop\ term\underline{)} \mid unop\underline{(}\ term\underline{)}$$

where *binop* is a binary operator and *unop* a unary operator. An example of a term is

$$(f(v) + (g(w) \times (x \div g(y))))$$

This term should correspond to one complex of the class. In figure 13.10 the object model is displayed. Each "node" represents a node in

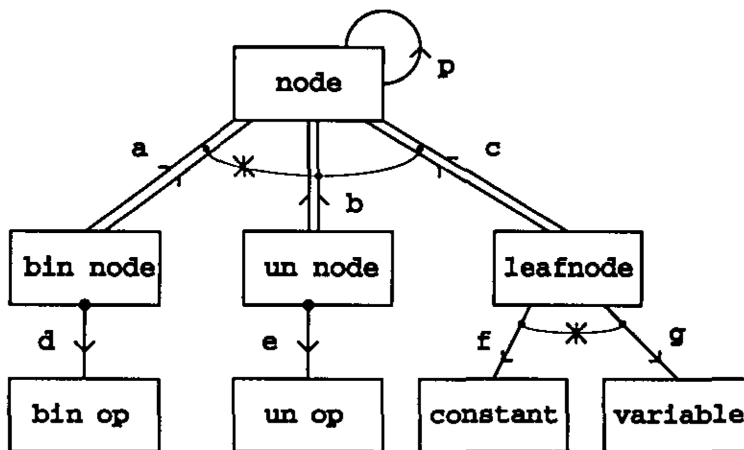

Figure 13.10: Syntax as complex class.

the parse tree of a term and relationship class *p* points to the predecessor of a node in the tree. In figure 13.11 the parse tree for the term above is displayed. A node has two branches, one branch or no branch which

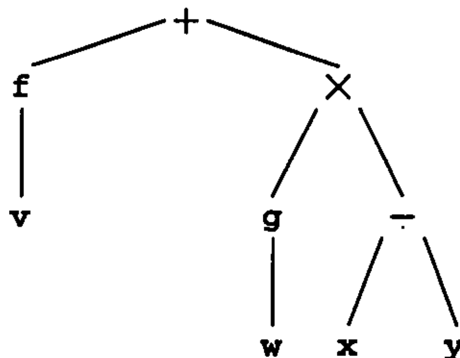

Figure 13.11: Parse tree.

corresponds to the different specializations of "node": "binnode", "unnode" and "leaf node" respectively. This is expressed by the following

additional constraints. (Note the range exclusion constraint.)

$$\forall n : node \bullet a^{-1}(n) \neq \{\} \Rightarrow size(p^{-1}(n)) = 2$$

$$\forall n : node \bullet b^{-1}(n) \neq \{\} \Rightarrow size(p^{-1}(n)) = 1$$

$$\forall n : node \bullet c^{-1}(n) \neq \{\} \Rightarrow size(p^{-1}(n)) = 0$$

Further we have to require that "node" and p form a tree as shown above. Finally we want to exclude that there are dangling nodes so

$$\forall x : node \bullet a^{-1} \neq \{\} \vee b^{-1} \neq \{\} \vee c^{-1} \neq \{\}$$

4. Representing history and future

We often start with an object model in which we represent the *actual* situation of a system, i.e. the simplexes represent the entities that are in the system and the relationships represent their actual properties. In the railway system for example the actual positions of the train and the switch were represented. In information systems we often consider the *history* of a process, which means that we store information objects that represent a part of a state of a target system in the past. Instead of information objects that refer to the past information systems have often information objects that refer to the *future*. These information objects are used for a *planning* for the future of the system. The future does not have to behave as planned, however the information objects represent what we think or wish to be the future. To transform an object model for an actual situation to one for the history or the future, we have to determine all the *time dependent* simplex classes, i.e. the simplex classes for which the simplexes may come and go during the course of the system. (Most simplex classes have this property). For all these simplex classes we introduce a new functional relationship connecting the simplexes to simplexes of a class called "time slot". The meaning of this relationship is that for each simplex the corresponding time slot indicates when the simplex was, is or will be "alive". If the time the simplex will die is not known then the upper bound of the time slot is not specified or is set to some sufficiently large number. Further we look for time dependent relationships. i.e. relationships that may change during the course of the system, independent of the life time of the simplexes they connect. For instance relationship l in figure 13.3 is time dependent because it may connect a "train" to different "tracks" during the life time of a "train". The way to solve the problem for the time dependent relationships is in fact the way to deal with relationships with properties: we introduce a new simplex class for these relationships and we connect it to the simplex classes that were connected by the original relationship. Further we connect the new simplex class to the "time slot" class. In figure 13.12 an example of this construction is displayed. Note that the time slots may refer to the history as well as to the future. If the original time-dependent relationship was functional, we obtain a

history

planning

time dependent

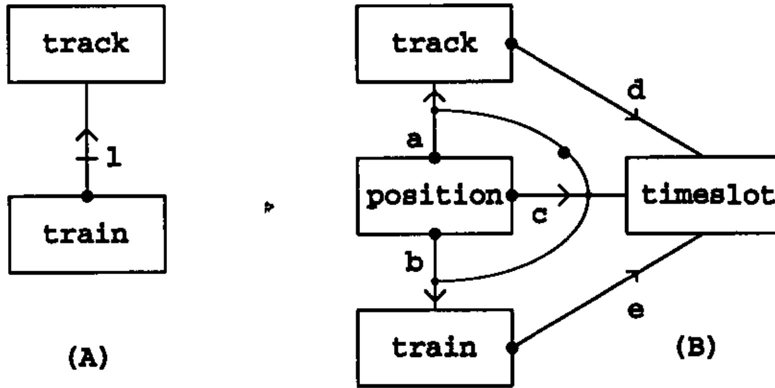

Figure 13.12: Time dependency.

constraint which states that the simplexes that replace the relationship may not be connected to more than one simplex of the domain simplex class of the original relationship at the same time. A second constraint states that the time slot of a "relationship" simplex is contained in the time slots of the original simplexes that are connected by the relationship simplex. In the example of figure 13.12 this means that the time slot of a "position" should be contained in the time slots of the "track" and the "train" to which it is connected by a and b respectively. To express these constraints formally we have to define a value type for time slots: $\mathbb{Q} \times \mathbb{Q}$, denoting the left and right bound of the time slot. Further we need two auxiliary functions:

$$intersect(x, y) := max(\pi_1(x), \pi_1(y)) \leq min(\pi_2(x), \pi_2(y)) : \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{B}$$

$$contain(x, y) := \pi_1(x) \geq \pi_1(y) \ \wedge \ \pi_2(x) \leq \pi_2(y) : \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{B}$$

The first constraint becomes:
$\forall x : position \bullet \forall y : position \bullet$

$$(x \neq y \ \wedge \ b(x) = b(y)) \Rightarrow \neg intersect(c(x), c(y))$$

And the second one:
$\forall x : position \bullet contain(c(x), e(b(x))) \ \wedge \ contain(c(x), d(a(x)))$
To guarantee that no two trains are on the same track, which is expressed by the injectivity of l in the first model, we require:
$\forall x,\ y : position \bullet$

$$a(x) = a(y) \ \wedge \ intersect(c(x), c(y)) \Rightarrow x = y.$$

5. Properties as values or relationships

Sometimes we have to choose between representing a property of a simplex in the representation of the simplex, i.e. in the value of the simplex, or as a relationship to another simplex class. Consider the example displayed in figure 13.13. Here we see that there are two total and bijective

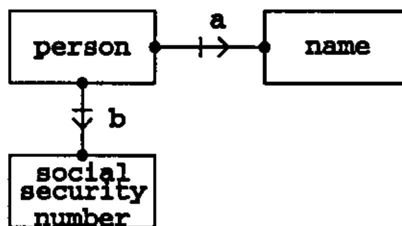

Figure 13.13: Properties as relationships.

relationships a and b. We could decide to delete for instance relationship a and give "person" the set of "names" as value type. If a would not have been total and bijective this was not possible (because if it was not total then there would have been persons without (known) names, if it was not functional a person could have more than one name and if it was not injective there could have been two persons with the same name). As a guideline we recommend to use the value of simplexes as less as possible, since this gives us most freedom in the specification stage. So it is recommended to use the construction as displayed in figure 13.13. In particular if there are more total bijective relationships (as in this case), it would be difficult to decide which one to incorporate in the value.

6. Aggregates

aggregate

Sometimes a simplex is just an *aggregate* of other simplexes. Consider the example displayed in figure 13.14. Here we see a simplex class "ad-

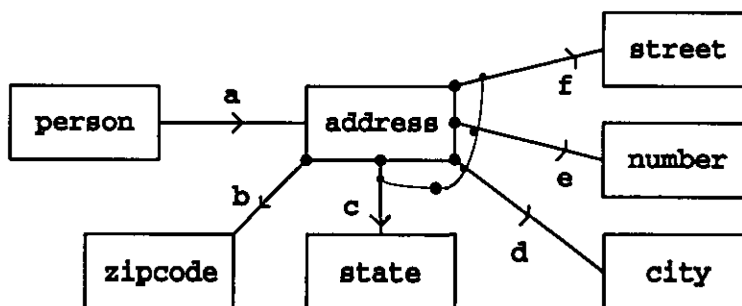

Figure 13.14: Aggregates.

dress" and intuitively we feel that an address *is* just a combination of a street, a number, a city and a state. So we could consider to define a complex class "address" which would have "street", "number", "city" and "state" as its body. With the constraint that there is only one of each simplex class in the complex, we would have been ready. However then we were neither be able to define relationships for "address" such as to the "zipcode", nor we could relate other simplexes such as "person" to "address". Besides that "address" is a distinct concept and for these reasons we introduce a simplex class "address". The value type of this simplex class is irrelevant because we can always identify an address by

its key constraint. Note that there is another domain key constraint for "address", not displayed in figure 13.14: the relationships $\{b, e\}$.

It is still useful to define a complex class, say "Address" which satisfies a tree constraint with "address" as root.

7. Use of inheritance constraints

Inheritance gives us the possibility to *differentiate* simplex classes into several simplex classes each with its own relationships. Consider for instance the example displayed in figure 13.15. Each vehicle has an

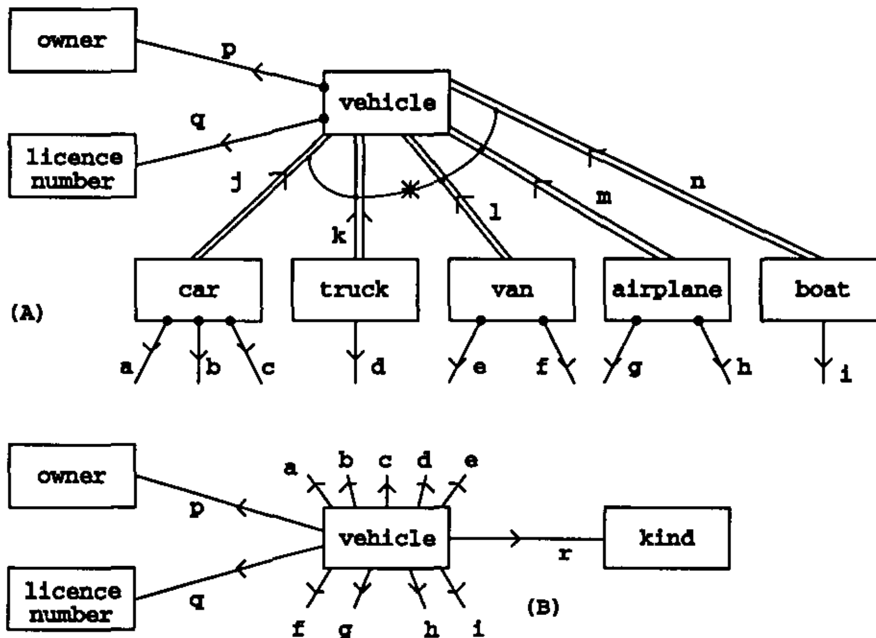

Figure 13.15: Use of inheritance.

"owner" and a "license number" however cars have other properties as airplanes or vans (displayed by relationships $a-i$). In solution (B) there is a simplex class "kind" that has a value type that represents the set

$$\{car,\ truck,\ van,\ airplane,\ boat\}$$

Further "vehicle" has all the relationships $a - i$. Although the diagram of (B) is more simple than the diagram of (A) there are much more constraints in the solution (B). Besides this, these constraints use values! For instance we would have to require that if a "vehicle" is not a "truck" then it should not have simplexes connected to it via relationship d:

$$\forall x : vehicle \bullet 'truck' \neq r(x) \Rightarrow d(x) = \{\}$$

So inheritance can be used to avoid non-total relationships (which is important for the *relational data model* as explained in the next section) and to avoid constraints that refer to simplex values. Note that we can easily use the relationships p and q for the specialized simplex classes

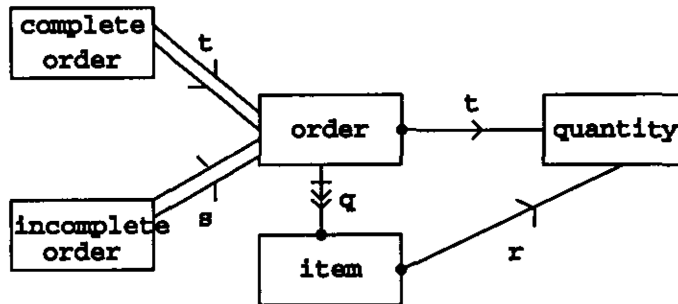

Figure 13.16: Temporal inconsistent orders

"car" ... "boat" by defining some auxiliary functions pj, $qj, \ldots, pn$, qn as follows:

$$pj(x) := p(j(x)) : car \Rightarrow \mathbb{F}(owner)$$

Another example of the use of inheritance is the treatment of temporal inconsistency. For example consider the class model of figure 13.16. Complete orders should satisfy the balance constraint

$$\forall o : order \bullet p(o) = sum(r(q(o))).$$

(Here *sum* is the function that adds the values in a set.) However not all orders are completely defined and therefore we require this for the complete ones only. For that reason we introduced the two simplex classes *complete_order* and *incomplete_order* and we require the constraint only for the complete ones:

$$\forall o : complete_order \bullet p(t(o)) = sum(r(q(t(o)))).$$

8. Derivable relationships

Sometimes we discover that a constraint is so strong that a relationship can be *derived* from the constraint. Consider the example displayed in figure 13.17. The example speaks for itself. The constraint that should

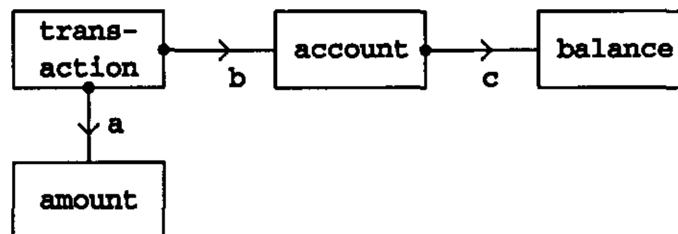

Figure 13.17: A derivable relationship.

hold is expressed using an auxiliary function:

$$s(t) := if\ t = \{\}\ then\ 0\ else\ a(pick(t)) + s(rest(t))\ fi :$$
$$\mathbb{F}(transaction) \Rightarrow \mathbb{Q}$$

It is assumed that the value types of "amount" and "balance" are equal

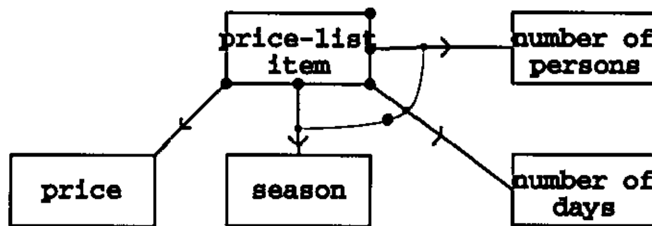

Figure 13.18: Price list

to $\mathbb{Q}$. The constraint becomes:

$$\forall y : account \bullet s(b^{-1}(y)) = c(y)$$

So the question is if it is not better to delete relationship c and to define an other auxiliary function:

$$balance(x) := s(b^{-1}(x)) : account \nrightarrow \mathbb{Q}$$

If there are no other relationships connected to the simplex class "balance" we could delete it too. Avoiding constraints means that we have less proof obligations in the specification phase!

9. Representing a finite function
Often we have to represent a finite function in an object model. For example a price list of a hotel is a function that lists the prices of hotel rooms depending on the season, the number of persons in the room and the number of days the room will be rented. The price is uniquely defined by these variables. In figure 13.18 we display a solution to this problem. In general there will be one simplex class for the (elements of) function and one for every argument of the function and one for the result. The simplex class that represent the function is related to the other simplex classes by means of total and functional relationships. The relationships with the variables as range simplex classes, form a (minimal) domain key. (In case the result of the function is compound, there may be more than one simplex class to represent the result.)

10. Object model of an actor model
Let an actor model be given. Now we will construct an object model that represents the actual state and the history of the actor model. This is useful in the context of the modeling of *monitoring information systems*. When we make a model of such an information system, we often start with a (complete) actor model of the target system and then we make an object model for the event history of this actor model. This object model may serve as a design for a database system in which the events of the actor model will be stored.

monitoring information systems

The method to construct an object model from an actor model proceeds along the following steps:

1. create for every processor a simplex class (with the same name),

2. create for every place two simplex class (one with the same name and one with the name decorated by '),
3. create a place called *Id* and a place called *Time*,
4. create for every connector a relationship (with the same name as the connector, if no name clash occurs) between the simplex class of the processor and the (undecorated) simplex class of the place that are connected by the connector (independent of the direction of the connector); the domain class of this relationship is the simplex class that represents the processor and the relationship is always functional and it is total if the processor will always consume or produce a token via this connector (in case of a name clash suitable names have to be chosen),
5. create for every (undecorated) place simplex class two total and functional relationships to the places *Id* and *Time*,
6. create a total and functional relationship from each undecorated place simplex class to the corresponding decorated simplex class,
7. for each simplex class n' that represents a place n we define:

$$sim(n') = com(CA(n)),$$

8. for the simplex classes *Id* and *Time* we define:

$$sim(Id) = ID \ \wedge \ sim(Time) = T,$$

   where ID is the set of identities and T the time domain,
9. for each other simplex class n define an arbitrary set for $sim(n)$,
10. create a complex class for each simplex class that represents a processor and let this complex class include all the simplex classes that are connected by a directed path of functional relationships; let the complex class satisfy a tree constraint with the processor simplex class as root.

The relationship between the actor model and the object model is as follows: whenever in the actor model a processor p executes, there will be a complex created (with simplex class p as root) and with four simplexes (belonging to a place) for every token that is consumed or produced during the execution of the processor: one simplex (of the undecorated simplex class) that denotes the token, one simplex (of the decorated simplex class) that denotes the complex of the token, one that denotes the identity and one that denotes the time stamp of the token.

Note that it is always possible to reconstruct the transition times: it is the maximum of the time stamps of the consumed tokens. To distinguish the consumed and produced tokens we have to inspect the actor model. (It is easy to modify the object model to incorporate the

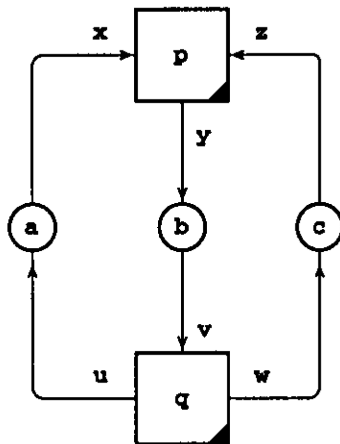

Figure 13.19: Actor model

necessary information as well.) We will illustrate this method by an example. The actor model represents a resource sharing system. In figure 13.19 the actor model is displayed and in figure 13.20 the corresponding object model. Note that the object model does not contain information about the way processors execute, so it cannot be used for forecasting or simulation, but only for monitoring the actor model. To derive the actual state of the actor net, we have to determine all the simplexes that represent tokens that do not occur as input tokens.

If we want to construct such a monitoring information system we have to modify the processors in the target system to enable them to produce the complexes defined above.

13.3 Transformations to other object frameworks

In this section we study methods to transform object models in our framework into object models in other frameworks and vice versa. The transformations are often only partial because some frameworks have specific requirements or lack some notions (for instance the notion of a complex). In the usual database terminology a framework defines a *schema* and a *set of instances* that belong to the schema. (Note that the term "schema" is used here in a different way as in the specification language.) In our framework, which we simply call the *object framework*, a *schema* is a class model plus the function *sim* that assigns a value type to each simplex class. The set of instances of a schema in the object framework, is the set of instances of the universal complex class.

schema
instance
object framework

For an arbitrary framework a schema plus its instances is called a *model* and is almost what we called an *object model.*

model
object model

Formally these transformations proceed along the following lines:

- first construct a (partial) function F from schemas in one framework into schemas of another framework,

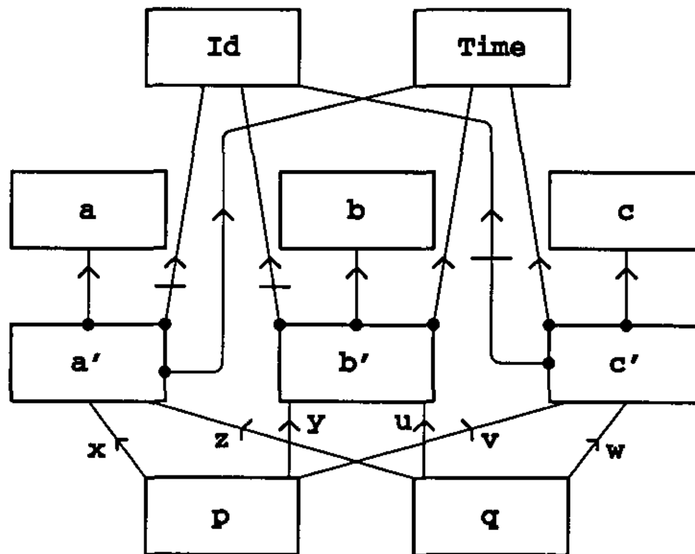

Figure 13.20: The corresponding object model

- next construct a (partial) *injective* function f that transforms an arbitrary instance a of a schema A in the first framework into an instance $f(a)$ of a schema $F(A)$ in the other framework.

information preservation

The fact that function f should be injective is very important: it means that f is *information preserving* because it has a (partial) inverse. So if it is possible to transform an instance a of framework A into an instance b of framework B we can reconstruct a from b. Note that in general the function f will depend on F and that there are several choices for F and f.

We only consider the cardinality and key constraints in these transformations, however it is possible to transform some other constraints as well.

relational data model

object oriented frameworks

Transformations of object frameworks are important for a number of reasons. First of all it is important to be able to communicate an object model to other persons who are more used to another object framework. Secondly it might be the case that an object model is already available in an other framework. Last but not least it might be necessary to implement an object model by means of a database management system that is based on an other object framework. Most database management systems used in practice are based on the *relational data model*, so this will be one of the frameworks we consider. Other frameworks are the *functional data model*, the *entity-relationship data model* and the *nested relational data model*, which is an extension of the relational data model. (Note that we called the entity-relationship data model "entity-relationship model" in part I.) There are several versions of each of these frameworks however we have chosen only one of them. There are some other object frameworks, but except for the *object oriented frameworks*, the ones presented here are the most important in practice.

The strategy we follow in this section is displayed in figure 13.21. Here "ERM", "OM", "FOM","RM" and "NRM" denote object models

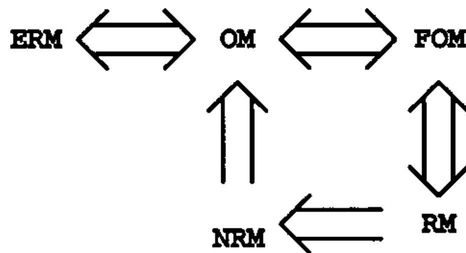

Figure 13.21: Strategy of transformations.

in the entity-relationship data model, (our) object framework, the functional data model, the relational data model and the nested relational data model, respectively. A double arrow head means that we consider a transformation in both directions. Since we are not expressing specifications but framework transformations we use here meta language.

1. Functional data model

The version of the functional data model we consider here is just a restriction of our framework in which only functional relationships are allowed. So the only problem is to get rid of the non-functional relationships. In practice most relationships are already functional (cf. for instance the examples in this chapter). However if there is a non-functional relationship then we can transform the schema as follows. In fact we define the function F from schemas in our object framework into schemas of the functional data model. We call an object model of the functional data model a *functional object model*.

functional object model

- if there is a non-functional, non-injective relationship use the "trick" we used to model relationships with properties, i.e. introduce a simplex class for every relationship that is non-functional and connect it to the domain and range class of the original relationship by total and functional relationships, having the new simplex class as their domain class, and these relationships have to form a domain key for the new simplex class,
- if a relationship is non-functional but injective, we may exchange the domain and range classes of this relationship and then we obtain a functional one (the proof of this statement is an exercise),
- the new simplex classes should have a value type, i.e. the function *sim* has to be defined for these classes; since there is a domain key constraint it does not really matter how we define these value types, because the simplexes are uniquely determined by their relationships, however we will use the next definition:

let r be a relationship in the functional object model that is replaced by simplex class d, then

$$sim(d) = sim(DM(r)) \times sim(RG(r)).$$

In figure 13.22 we see examples of these transformations. Note that p is

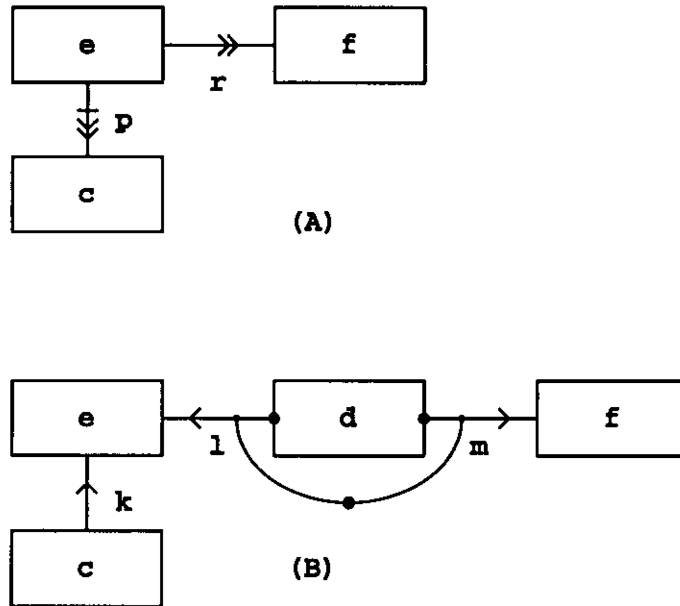

Figure 13.22: Transformation to a functional model.

replaced by k and r by l, d and m. Now we have seen how to transform the schema. However we also have to transform instances of our schema into instances of a schema of the functional data model. We only consider universal complex classes, because the functional data model does not have the notion of complex classes. So we consider an arbitrary universal complex a of an object model in the object framework and we will transform it into a universal complex b of the functional data model. We will call the schemas respectively A and B, so $B = F(A)$. The transformation (f) proceeds along the following lines:

1. for each simplex class n of schema A we have $b(n) = a(n)$,
2. for each functional relationship r of model A the same relationship occurs in schema B and $b(r) = a(r)$ and it has to satisfy the same cardinality constraints,
3. for each non-functional but injective relationship p of schema A there is a relationship k of schema B with:

$$b(k) = \{(x, y) \mid (y, x) \in a(p)\}$$

if p is total then k is surjective and if p is surjective then k should be total,

4. for each non-functional, non-injective relationship r of schema A there is a simplex class d and two relationship classes l and m such that $DM_B(l) = DM_B(m) = d$, $RG_B(l) = DM_A(r)$ and $RG_B(m) = RG_A(r)$;
$b(d) = a(r) \ \wedge \ b(l) = \{((x, y), x) \mid (x, y) \in a(r)\} \ \wedge$
$$b(m) = \{((x, y), y) \mid (x, y) \in a(r)\}.$$

Note that $b(d) = a(r)$ is justified by the choice of $sim(d)$. It is clear that F is not injective. To verify this note that models (A) and (B) of figure 13.22 are mapped both to model (B). However f is injective. (The proof of this assertion is an exercise.) In fact we may consider the function F as a reduction to a *normal form* of the object framework. Therefore this framework is sometimes called the *irreducible data model*, while our object framework is sometimes called in literature, the *functional data model*, because the binary relationships may be considered as (set-valued) functions.

normal form

irreducible data model

To transform an object model defined as a functional data model into one in our framework, we have to do nothing because it is already an object model in the object framework i.e. the functions F and f are the identities.

2. Relational data model

In order to describe the transformations we need a definition of the relation data model. We call an object model of the relational data model a *relational model*. A relational model is defined by a *relational schema* and the set of *instances* of this schema.

relational model

relational instance

Definition 13.1 A *relational schema* is a 5-tuple:

relational schema

$$(T,\ A,\ \alpha,\ \beta,\ \gamma)$$

where T and A are mutually disjoint sets and:

- T is the set of *relation* names,
- A is the set of *attribute* names,
- $\alpha : T \rightarrow \mathbb{P}(A)$ assigns to every relation name a set of attributes; we require that:
$$\forall t_1, t_2 \in T : t_1 \neq t_2 \Rightarrow \alpha(t_1) \cap \alpha(t_2) = \emptyset,$$
- β is a function that assigns to every attribute a set called an *attribute domain* (note that $\text{dom}(\beta) = A$),

attribute domain

- $\gamma : T \rightarrow \mathbb{P}(A)$ assigns to every relation one *primary key*, which is a subset of the attributes assigned by α, such that
$$\forall t \in T : \gamma(t) \subset \alpha(t) \ \wedge \ \gamma(t) \neq \emptyset.$$

primary key

□

(The terminology used here is the usual one for the relational data model, however note that some of the terms have a slightly different meaning in the rest of this book.) Note that if the attribute names of two relations are not disjoint we can make them disjoint by combining their names with the names of the relations in which they occur. In the following table we display a relational schema. The code "y" means that the attribute is part of the primary key of the relation and "n" means that it is not part of it the primary key.

relation	attribute	domain	key
r_1	a_1	A_1	n
	a_2	A_1	y
	a_3	A_2	y
r_2	a_4	A_2	y
	a_5	A_3	y
	a_6	A_3	n
r_3	a_7	A_3	y
	a_8	A_4	y
	a_9	A_4	n

Next we have to define instances of a relational schema, which are comparable to our universal complex class (cf. 9.2). The set of instances is defined by:

instance

Definition 13.2 Let a relational schema be given. An *instance* of a relation database schema is a function b with $dom(b) = T$ and

$$\forall t \in T : b(t) \subset \Pi(\beta \restriction \alpha(t)) \ \wedge \ (\forall x, y \in b(t) : x \restriction \gamma(t) = y \restriction \gamma(t) \Rightarrow x = y).$$

□

relation
tuple
functional dependencies
multi-valued dependencies
referential integrity

For $t \in T$ and an instance b the set $b(t)$ is a set of functions, called a *relation*, with a common domain $\alpha(t)$. These functions are called *tuples.* The primary key identifies a tuple in an instance. (Note that it is a minimal key, because we have no other keys defined.) The relational data model has several standard constraints: *functional dependencies*, *multi-valued dependencies* and *referential integrity*. Referential integrity is equivalent with our surjectivity constraint. A key constraint is the most important example of a functional dependency and the multi-valued dependencies do not have an equivalent in our framework; therefore we do not consider them.

Note that the concept of a complex is not existent in the relational data model so we have to restrict the transformation to the universal complex class. For the relational data model exist two *query languages*: the *relational algebra* and the *tuple calculus*. In the next part we show how the relational algebra can be expressed in the specification language.

First we consider transformations from relational models and object models. In figure 13.23 we see the corresponding object model of the

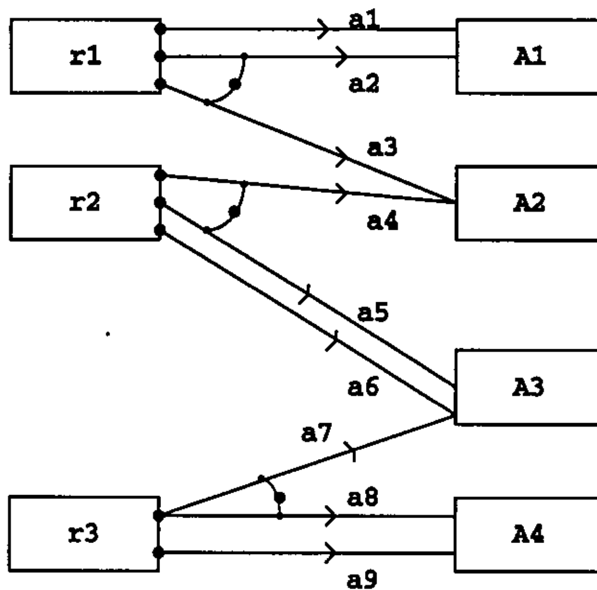

Figure 13.23: Transformation of a relational schema.

relational schema in the table above. First we define the function F that maps a relational schema A into a schema for the object framework, called B.

1. $SN = T \cup rng(\beta)$, so every relation and every attribute domain become simplex classes
2. $RN = A$, so all attributes become relationships,
3. $\forall a \in RN, t \in T : DM(a) = t \Leftrightarrow a \in \alpha(t)$ (note that t is uniquely determined),
4. $\forall a \in RN : RG(a) = \beta(a)$,
5. $\forall t \in T : DK(t) = \{\gamma(t)\}$,
6. $\forall t \in T : sim(t) = \Pi(\beta \upharpoonright \gamma(t))$,
7. $\forall r \in rng(\beta) : sim(r) = \beta(r)$.

Next we consider the transformations on the instance level. The transformation f maps an instance a of A into a universal complex b of $F(A)$.

1. $\forall t \in T : b(t) = \{x \upharpoonright \gamma(t) \mid x \in a(t)\}$, so all primary keys of a relation form a set of simplexes of a simplex class with the same name as the relation,
2. $\forall d \in rng(\beta) : b(d) = \{x \upharpoonright r \mid \exists t \in T : x \in a(t) \wedge r \in \alpha(t) \wedge \beta(r) = d\}$, so all attribute values of an attribute domain that occur in

a form the simplexes of a simplex class with the name of the attribute domain,

3. $\forall r \in RN : b(r) = \{(x, y) \mid \exists z \in a(DM(r)) : x = z \restriction \gamma(DM(r)) \land y \in x(r)\}$, note that $b(r)$ is total and functional because of the property of the ' primary keys of the relational data model

It is clear that these rules define a function. However we still have to prove that this function f is injective.

Lemma 13.1 The function f defined by the three rules above is *injective.*

Proof. Let a_1 and a_2 be two instances of the relational schema A and suppose $f(a_1) = f(a_2) = b$. We have to prove that $a_1 = a_2$. Take an arbitrary $t \in T$. We have (by rule 1):

$$\{x \restriction \gamma(t) \mid x \in a_1(t)\} = \{x \restriction \gamma(t) \mid x \in a_2(t)\}.$$

Let $x_1 \in a_1(t)$ and $x_2 \in a_2(t)$ such that $x_1 \restriction \gamma(t) = x_2 \restriction \gamma(t)$. (Note that this does not imply $x_1 = x_2$, since x_1 and x_2 belong to different instances.) Then, for all $r \in \alpha(t)$ (by rule 3):

$$(x_1 \restriction \gamma(t), x_1(r)) \in b(r) \ \land \ (x_2 \restriction \gamma(t), x_2(r)) \in b(r).$$

However r (as relationship) is functional and therefore $x_1(r) = x_2(r)$. This proves that $x_1 = x_2$. So we have proven that $a_1 = a_2$.
□

We continue with the transformation the other way round. So we start with a schema A of an object model and we first transform its schema into the schema B of a relational model (function F) and afterwards we define the transformation f that maps an instance of a universal complex class into an instance of the relational schema. However we first transform the object model into a functional model a seen before. So we assume that A is a functional model and in addition we assume that all relationships are total, because this avoids the problem of nil values in the relational model. Note that F is partial now! Transformation F is defined by:

1. $T = rng(DM)$, so only simplex classes with "properties" become relations,

2. $A = RN \cup \{t' \mid t \in T\}$, so all relationships become attributes and for each relation t there is one new attribute t' (assume primes where not used in names of A),

3. $\forall t \in T : \alpha(t) = \{t'\} \cup DM^{-1}(t)$,

4. $\forall r \in RN : \beta(r) = sim(RG(r))$,

5. $\forall t \in T : \beta(t') = sim(t)$,

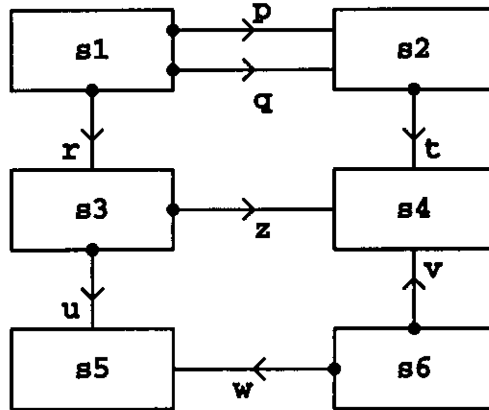

Figure 13.24: Transformation of a functional model into a relational model.

6. $\forall t \in T : \gamma(t) = \{t'\}$, so the new attributes form the primary keys.

In figure 13.24 we see a schema of a functional model and in the following table we see its transformation into a relational model.

relation	attribute	domain	key
s_1	s'_1	$sim(s_1)$	y
	p	$sim(s_2)$	n
	q	$sim(s_2)$	n
	r	$sim(s_3)$	n
s_2	s'_2	$sim(s_2)$	y
	t	$sim(s_4)$	n
s_3	s'_3	$sim(s_3)$	y
	u	$sim(s_5)$	n
	z	$sim(s_4)$	n
s_6	s'_6	$sim(s_6)$	y
	v	$sim(s_4)$	n
	w	$sim(s_5)$	n

The next step is the definition of f. Let a universal complex a be given, instance $b = f(a)$ should satisfy:

- $\forall t \in T : \{x(t') \mid x \in b(t)\} = a(t)$,
- $\forall t \in T : \forall r \in \alpha(t) :$,

$$\{(x(t'), x(r)) \mid x \in b(t)\} = a(r).$$

It is not a priori evident that these rules determine b uniquely.

Lemma 13.2 The two rules above define a function.

Proof. Let a be given. We have to show that a determines only one b. Suppose that b_1 and b_2 satisfy the rules above. We will show that

$b_1 = b_2$. Choose a $t \in T$. Let $x_1 \in b_1(t)$. Then $x_1(t') \in a(t)$ which implies

$$\exists x_2 \in b_2(t) : x_2(t') = x_1(t').$$

Further $\forall r \in \alpha(t) : (x_1(t'), x_1(r)) \in a(t) \Rightarrow$

$$\exists x_3 \in b_2 : (x_3(t'), x_3(r)) = (x_1(t'), x_1(r)).$$

The fact that t' is a primary key for $b_2(t)$ implies that

$$\forall r \in \alpha(t) : x_3(t') = x_2(t') = x_1(t') \ \wedge \ x_3(r) = x_2(r) = x_1(r).$$

This implies that $x_1 = x_2$ and so $x_1 \in b_2(t)$. Therefore $b_1 = b_2$.
□

Finally we have to show that f is injective.

Lemma 13.3 The function f defined by the rules above is injective.

Proof. The proof is an immediate consequence of the specification of f: if a_1 and a_2 are two universal complexes with $f(a_1) = f(a_2)$ then the specifications for a_1 and a_2 by the rules above are identical (namely $b = f(a_1) = f(a_2)$ and so $a_1 = a_2$.
□

Note that if we transform a relational schema into an object schema and afterwards this object schema into a relational schema, then the last schema is identical to the first one except that each relation has one extra attribute.

3. Entity-relationship data model

As in the case of the relational data model we start with a definition of this framework.

entity-relationship schema

Definition 13.3 A *entity-relationship schema* is a 7-tuple $(E, R, A, \alpha, \beta, \gamma, \delta)$ where:

- E is the set of *entities*,
- R is the set of *relationships*,
- A is the set of *attributes*,
- $\alpha : E \to \mathbb{P}(A)$ assigns to every entity a set of attributes,
- β is a function that assigns to every attribute a set called the *attribute domain*,
- $\gamma : E \to \mathbb{P}(A)$ assigns to every entity a *primary key*, such that (margin: **primary key**)

$$\forall e \in E : \gamma(e) \subset \alpha(e),$$

- $\delta : R \to \mathbb{P}(E)$, which assigns to every relationship a set of entities.

□

This framework does not have the notion of complex classes either so we restrict ourselves to the universal complex class as before. Normally the entity-relationship data model is used as an aid to define a relational schema and in that case one does not have to define instances of entity-relationship schemas. However we transform an *entity-relationship schema* into a schema of an object model and so we define indirectly instances for entity-relationship schemas as instances of the universal complex class of the corresponding object model! Therefore we do not have to specify the function f that transforms instances. Note that the term "relationship" is used here different, and therefore we will call it an "er-relationship". The transformation proceeds along the following lines:

1. $SN = E \cup R \cup A$,
2. $RN = \bigcup_{e \in E}\{(e,a) \mid a \in \alpha(e)\} \cup \bigcup_{r \in R}\{(r,e) \mid e \in \delta(r)\}$,
3. $\forall (x,y) \in RN : DM((x,y)) = x \ \wedge \ RG((x,y)) = y$,
4. $\forall e \in SN \cap E : DK(e) = \{\{(e,a) \mid a \in \gamma(e)\}\}$,
5. $\forall r \in RN \cap R : DK(r) = \{\{(r,e) \mid e \in \delta(r)\}\}$,
6. all relationships of the object model are total and functional.

In figure 13.25 we display an entity-relationship schema and its transformation into an object model.

It is also possible to transform schemas of object models into entity-relationship schemas. An *isomorphic* transformation is possible if the object model has some structural properties. In this case the graphs of the object model and the entity-relationship schema are isomorph, i.e. there is bijective mapping between the nodes and edges and in fact this is the classification of simplex classes introduced in section ??. The structural properties can be expressed using the concept of *level*. A simplex class n has level 0 if and only if $n \notin rng(DM)$, i.e. n has no properties. A simplex class n has level k if and only if it all simplexes in $\{RG(r) \mid DM(r) = n\}$ have level $k-1$. (Note that in an arbitrary object model not all simplex classes have a level.) We will use the notation $level(n)$ for the level of a simplex class n. The conditions for an isomorphic transformation are:

- all simplex classes have level 0, 1 or 2,
- all relationship classes are total and functional,
- no two relationship classes have the same domain and range classes.

The simplex classes of level 2 become er-relationships, the simplex classes of level 1 become entities and the simplex classes of level 0 become attributes. (We call the last ones *attribute simplex classes*.)

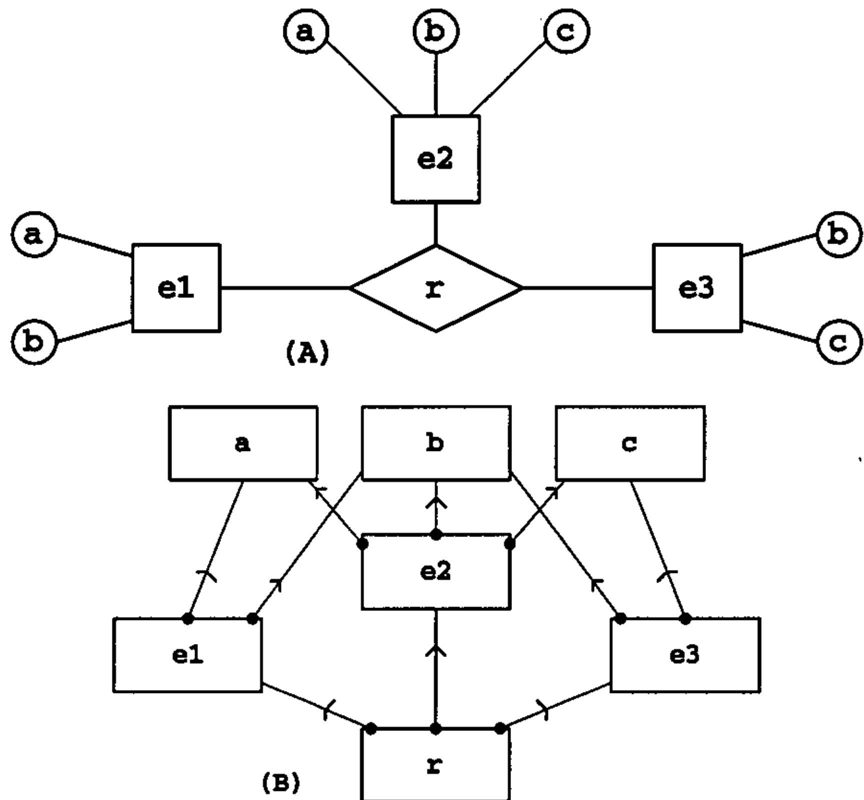

Figure 13.25: Transformation of an entity-relationship schema.

Further we require that every simplex class has one domain key constraint consisting of relationship classes that have attribute simplex classes as their range classes. Let such an object model be given, then the corresponding entity-relationship schema is:

1. $A = \{r \in RN \mid level(RG(r)) = 0\}$, so the relationship classes that have "attributes" as range class become attributes themselves
2. $E = SN \backslash \{n \in SN \mid level(n) = 0\}$, so the attribute simplex classes disappear
3. $R = RN \backslash A$,
4. $\forall e \in E : \alpha(e) = \{r \in A \mid DM(r) = e\}$,
5. $\forall a \in A : \beta(a) = sim(RG(a))$,
6. $\forall r \in R : \delta(r) = \{DM(r), RG(r)\}$,
7. $\forall e \in E : \gamma(e) \in DK(e)$.

Note that er-relationships have no direction, which causes no problem because an er-relationship is never connected to the same entity twice.

In the general case, where some simplex classes do not have to have such a domain key constraint, we have to create an extra attribute for each entity with the same value domain as the value type of the simplex class. (This construction is an exercise.) As stated in part I, the object framework may be considered as an extension of the binary version of the entity-relationship model, which means that $\forall r \in R : \#(R(r)) = 2$. This is what we have shown here.

4. Nested relational data model

The nested relational data model is a generalization of the relational data model. This framework enables us to model "non-atomic" attributes in a more easy way. We start with an example in the following table.

Order	Item	Total	Supplier	Price	Quantity
o_1	i_1	100	A	3	50
			B	5	30
			C	4	20
	i_2	50	B	7	20
			D	6	30
o_2	i_3	60	B	8	40
			E	10	20
	i_4	80	A	10	50
			D	12	30

To indicate that a "nest" (in fact a row) of attributes may be repeated we use curly brackets, so the attributes of the table above can be coded by:

$$\{(Order,\ \{(Item,\ Total,\ \{(Supplier,\ Price,\ Quantity)\})\})\}$$

In this table we see only one nested relation. There is no need for more than one nested relation because we can combine two of them to form one. For instance two non-nested relations T_1 and T_2 with attribute sets $\{A, B, C\}$ and $\{D, E, F\}$ respectively, can be combined into one nested relation, coded by:

$$\{(\{(A,\ B,\ C)\}, \{(D, E\ F)\})\}$$

So a relational model can be transformed in an information preserving way, into a nested relational model (the proof is an exercise). We start with the definition of a *nested relational schema* and afterwards we give the definition of an instance of such a schema.

nested relational schema

Definition 13.4 A *nested relational schema* is a 3-tuple:

$$(A, \beta, T)$$

where:

- A is a finite set of attributes,
- β is s function that assigns to every attribute a set, called *attribute domain*,
- T is an *attribute nest*; attribute nests are defined using a syntax, by:
  - *AttributeNest* ::= $\underline{\{}$*Nest*$\underline{\}}$,
  - *Nest* ::= *Attribute* | $\underline{(}$*Nest*$\underline{,}$*Nest*$\underline{)}$ | $\underline{\{}$*Nest*$\underline{\}}$,
  - *Attribute* $\in A$,
  - no attribute may occur twice in an attribute nest.

  An *attribute nest* is a set that can be represented by an *AttributeNest*.

□

Next the set of instances of a nested relation schema is defined.

Definition 13.5 Let a nested relation schema be given. A *instance* of such a schema is defined recursively using the set X of all *sub-attribute nests* of T: X is the set of all attribute nests that are represented by a sub-string of T. The set of all instances of x, where $x \in X$, is denoted by $I(x)$. The function I is (recursively) defined by:

- $\forall x \in X \cap A : I(x) = \beta(t)$,
- $\forall x, x_1, \ldots, x_n \in X : x = (x_1, \ldots, x_n) \Rightarrow I(x) = I(x_1) \times \ldots \times I(x_n)$,
- $\forall x, y \in X : x = \{y\} \Rightarrow I(x) = I\!F(I(y))$.

□

So in the table above the instance
$\{(o_1, \{(i_1, 100, \{(A, 3, 50), (B, 5, 30), (C, 4, 20)\}),$
$(i_2, 50, \{(B, 7, 20), (D, 6, 30)\})\}),$
$(o_2, \{(i_3, 60, \{(B, 8, 40), (E, 10, 20)\}),$
$(i_4, 80, \{(A, 10, 50), (D, 12, 30)\})\})\}$
is displayed.

As noted before, the relation data model can be transformed into the nested relational data model. We have seen how to transform an object model of our framework into the relational data model. The final step to close the circle is to show how a model in the nested relational data model can be transformed into our framework. Then we have also shown that a nested relational model can be transformed into a relational model. (Of course it is possible to give a more direct transformation than we present here.)

We first consider the transformation of a nested relational schema into a schema of an object model. It proceeds along the following lines:

1. create a simplex class for every sub-attribute nest in X, including the attributes themselves, and give them a suitable name (use for example the elements of X as names),

2. create for every simplex class with a name of the form $\{x\}$, a relationship class r that is total, functional and that satisfies:

$$DM(r) = \{x\} \wedge RG(r) = x$$

,

3. create for every simplex class with a name of the form $(x_1, \ldots, x_n)$ relationship classes $r_1, \ldots, r_n$ which are total and functional and that satisfy:

$$\forall i \in \{i, \ldots, n\} : DM(r_i) = (x_1, \ldots, x_n) \wedge RG(r_i) = x_i,$$

4. $\forall n \in SN : DK(n) = \{DM^{-1}(n)\}$,

5. for each simplex class n that represents attributes: $sim(n) = \beta(n)$, for the others we may chose the value types arbitrarily.

In figure 13.26 the transformation of the example above is displayed. (Note that we have shortened the names of attributes.) In fact the simplex classes with numbers 3 and 5 are redundant. In figure 13.27 we transformed (injective) the object model in order to obtain a more simple one with the same information. (The proof of this statement is an exercise.)

On the instance level we can define an injective function f that transforms an instance of a nested relational schema into a universal complex that satisfies a tree constraint (with the simplex class that represents the whole attribute nest as root). This construction proceeds along the same lines as the schema transformation. (The specification of this function is an exercise.)

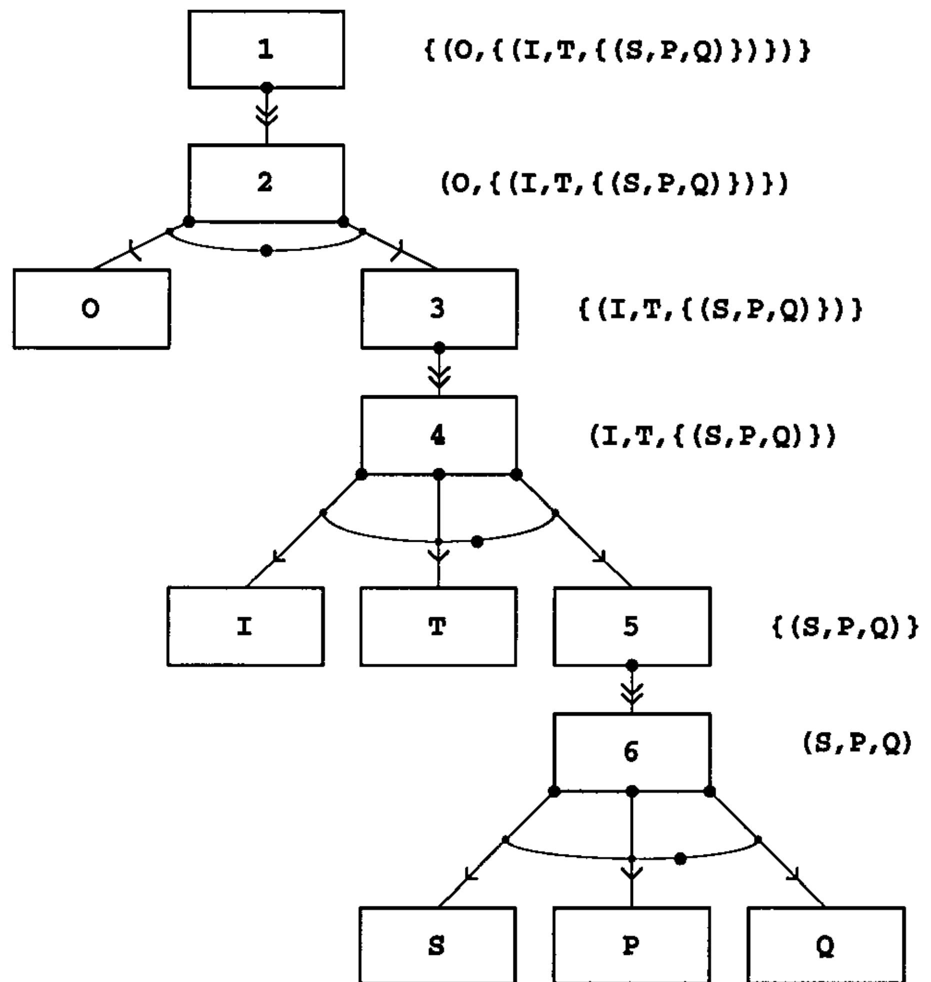

Figure 13.26: Transformation of a nested relational schema.

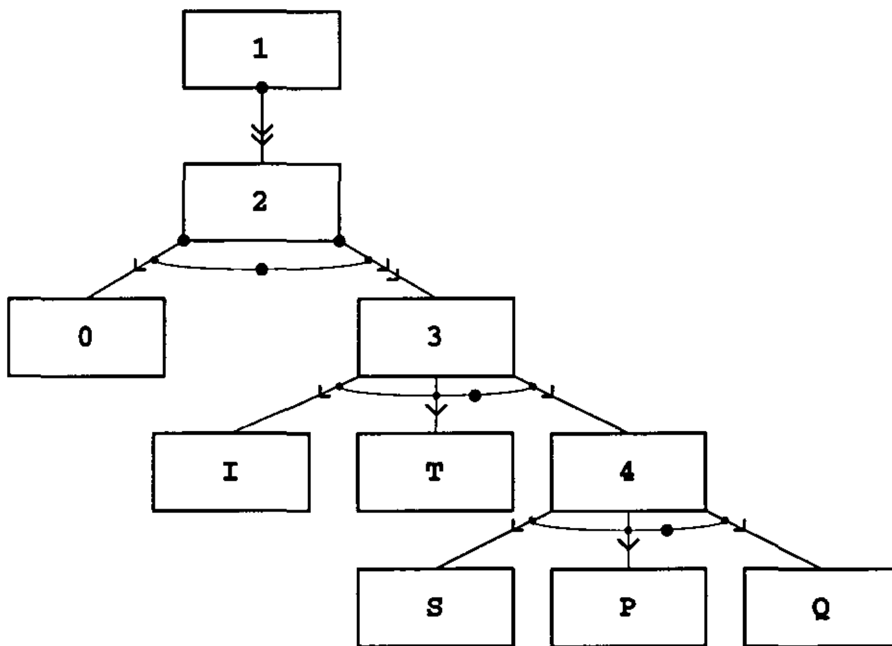

Figure 13.27: Reduction of the object model of a nested relational schema.

Chapter 14

Object oriented Modeling

As said before, we consider *object oriented modeling* as a method to construct a complete actor model in an integrated way. The method uses a specific paradigm of a system (the object oriented paradigm), which can easily be mapped onto our frameworks. This paradigm is quite informal, and so there are many ways to formalize it. We will start with the main ideas of the object oriented paradigm and afterwards we will show how these ideas can be incorporated in our frameworks.

object oriented modeling

The basic idea of the object oriented paradigm is that there are classes of *active objects*. (Note that we use the term "object" a bit different here.) Each object has a (structured) value and it may have *knowledge* of other objects. Each object has a *life cycle* that starts with its *birth* and that ends with its *death*, also called *creation* and *deletion* of the object. During its life an object can change its value, and it can exchange *messages* with other objects of the same or different classes. The structure of the life cycle is the same for all objects of a specific class and the type of communication to other objects is also determined by the class of the object. A system is considered as a "dynamic set" of objects, which means that at each moment in time there is for each object class a finite number of objects, each with a particular value and some particular knowledge of other objects, and that there are pending messages destined for specific objects. This "dynamic set" changes over time because the objects may change their values, receive messages and send messages. A change of value of an object may be triggered by a message but this is not necessarily the case, it may also change its value autonomously. The operations that change the value and the knowledge of an object are usually called *methods*, they are specific for an object class. The value of an object and the knowledge it has of other objects at a particular time can be considered as the *state* of the object. There are several situations that should be avoided, such as that there exist messages for objects that died already or that two or more objects are waiting for messages of each other (dead lock).

active objects

knowledge

life cycle

message

method

The idea of object oriented modeling is that a systems engineer can define an object class completely in *isolation*, i.e. without knowledge of other object classes. Object oriented *languages* have facilities to use

inheritance relationships between object classes, which may decrease the modeling effort of a system.

If we compare the object oriented paradigm with our three frameworks we see an important difference: objects in our framework are *passive* components of a system and actors are *active* components but actors are fixed, i.e. there are no births and deaths of actors! So it is not immediately clear if we should identify the "object oriented" objects with our objects or with our actors. To distinguish the object oriented objects from ours, we will call them *o-objects*. It turns out that we have for each *o-object class* a *complex class* and an *actor*.
Object oriented modeling proceeds along the following lines:

o-complex class

- For each o-object class there is a complex class that satisfies a tree constraint and the root simplex identifies the o-object (the type is ID), we call it an *o-complex class*.

o-actor

- For each o-object class there is one actor that represents the life cycle of the o-object, we call it an *o-actor*.

state machine

- Every o-actor has internally the structure of a *state machine*, i.e. each processor is connected to at most one input and one output place within the actor. A processor may have other input and output connectors that are connected to the connectors of the o-actor.
- The input and output connectors of an o-actor are used for the exchange of messages with context actors or with o-actors.

m-complex class

- For each connector of an actor there is a complex class that represents a message type. We call it an *m-complex class*.
- A life cycle of an o-object can only start with a message from outside. There may be live o-objects in the initial state, so an o-actor does not have to have an input connector for life cycle creation. A message that starts the life of an o-object may be sent by a context actor or by an o-actor. In the last case it may be the same o-actor, which means that o-objects of one class may create new ones from the same class. A life cycle may end or may continue for ever.

knowledge simplexes

value simplexes

- The o-complexes have two kinds of simplex classes: simplex classes that are root simplex classes of other o-complex classes and other simplex classes. The first kind of simplexes represent the knowledge of another o-object: if an o-complex contains a root simplex of another o-complex it means that it knows of the existence of the other o-complex. We call them *knowledge simplexes* (k-simplex for short). The other simplexes in an o-complex denote the value of the o-object. We call them *value simplexes* (v-simplex for short). In many cases the k-simplexes will only have relationships with the root simplex in an o-complex.

- Newly created o-complexes obtain their identity (in the root simplex) from the token containing the message that initiated their life. So at the start the identity of the o-object and the identity of the token that contains it, are equal. During the life of the o-object the identity of the containing token changes but the identity of the o-object remains the same.

- An m-object contains the identity of the sending o-object (the return address) , and in some cases also of the receiving o-object. There are however cases in which the receiving object is not known because the message may be handled by any o-object of the addressed class.

- The *state* of an o-object is determined by the value in the o-complex plus the place in the o-actor where the o-complex resides. So in fact the *token* that carries the o-complex represents the state of the o-object, since a token contains the place information. (Note that the places in an o-actor are *stages* in the life cycle of the o-object.)

state

stage

- The processors inside an o-actor perform the state changes. They may be triggered by an incoming message and they may produce an outgoing message. They may be considered as the *methods* of an o-object.

o-object method

- The communication between two o-objects needs a *protocol*, i.e. a token exchange pattern. There are two kinds of communication. The first kind of communication concerns the creation of an o-object by some other o-object. The second kind concerns a *client-server* behavior. Here one o-object (the *client*) asks a service of another o-object (the *server*). The server may ask another o-object to perform a part of this service. So the server may behave as a client as well and one request for service may create a cascade of requests.

  In most cases a message is answered by an other message. A simple protocol is that an o-object has at most one message pending at a time. So after it has sent a message it may perform *internal steps* (i.e. steps without sending messages) only until it receives an answer from the receiving o-object.

protocol

client-server

Note that what is called an o-object here, can also be considered as a *transaction*. A transaction in a database system for instance, also has a life cycle and it may initiate other transactions and it may wait for reactions of other transactions. So the object oriented paradigm is applicable to transaction processing systems in a natural way.

transaction

As mentioned before the object oriented modeling method develops a model by considering one o-object class at a time. This means that for each class the following activities have to be carried out:

1. Define an o-complex class, i.e. an object model.

2. Define an o-actor, i.e. an incomplete actor model. The places inside the o-actor mark the stages in the o-objects life cycle.

3. For each connector of this o-actor an m-complex has to be defined, if it was not defined before for another class.

4. Specify suitable value types for the complex classes involved.

5. Specify the processor relations for the o-actor. Usually these will be functional and the corresponding functions are called methods.

Note that we did not use hierarchy of actors in this method. Of course it can be used to "hide" a part of the life cycle in an actor. This actor has internally also the state machine structure and has one input and one output connector. So it can be considered as a processor, because it behaves as such.

Stores can be used as well, but they do not fit very well in the object oriented paradigm. In practice it is good to start with an overall actor model of the system in which the context actors and o-actors are displayed. The use of inheritance that is supported by most object oriented languages is not directly translatable to our framework. Of course we have polymorphic functions, and type variables that give us the possibility of reusing already defined constructions, but we have no inheritance relationships between actors or complex classes.

The question that remains is: how to find the o-object classes? There is no "waterproof" answer to this question. If the paradigm is carried through too far, every "thing" is considered as an o-object and this means that we get many o-object classes with a simple structure but with many complicated interactions between objects of these classes. For example if we consider a library system and we consider each simplex of an object model in our framework as an o-object then we have o-object classes for books, for authors, for publishers, for dates etc. This would mean that if a library user wants to ask a question about a book, he has to send a message to the book and then the book has to send a message to the author(s) and to the publisher and to the date (of publication). Of course this can be modeled in this way, but it is not a natural way of modeling and certainly not a simple model. A good approach is to consider only those entities in the real world as o-objects that behave as o-objects, i.e. they have their own life cycle and they communicate with other entities. These entities should belong to classes, which means that there could be more instances of the class at the same time. So in fact there is a simple answer to the question: if real world items can be identified as o-objects in a natural way, then they should be modeled in this way, otherwise they should be modeled as simplexes and be incorporated in o-complexes or m-complexes.

We will illustrate the object oriented modeling method by a small example. Consider a jobshop, i.e. a factory that has resources that can be used to perform tasks and clients send jobs consisting of one or more tasks to the factory. (Note that this model is a simplification of the

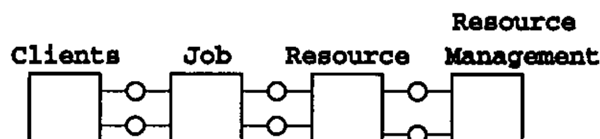

Figure 14.1: Jobshop: top level

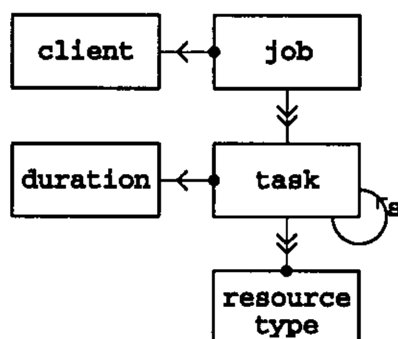

Figure 14.2: o-complex class for Job: ShopOrder

factory example considered in chapter 13.) In figure 14.1 we show the top level of the system. There are two context actors, called *Clients* and *ResourceManagement*. The first one is sending jobs to the jobshop and the second one is adding new resources, taking resources out and is reserving time for maintenance of resources. We have connected them by channels, however in this early stage of development it is not sure via how many connectors the o-objects will communicate. The context actors are not considered in detail so we concentrate on the two o-object classes: *Job* and *Resource*. We start with *Job*. The o-complex class for *Job* is displayed in figure 14.2. The simplex class *job* is the root of the complex. Tasks have an ordering, which is expressed by the functional relationship s that assigns to a task its successor. It needs a constraint as we have seen in section 13.2. We call this complex class: *ShopOrder*. The next step is the o-actor for *Job*. It is displayed in figure 14.3. Processor t_1 creates a new job from a message of a client and t_4 deletes a job. All processors are functional and complete, but only t_1 and t_2 are total. The other processors have preconditions: t_3 selects pairs of input tokens that belong to the same job and t_4 and t_5 select on the existence of unfinished tasks in the jobs. Note that from each job only one task at a time can be executed. However several jobs may be processed concurrently. The m-complex classes for the connectors are displayed in figure 14.4. All of them satisfy a tree constraint with *client* or *job* as root. The m-complex class for connector c_1 is almost the same as the o-complex class. The only difference is the job identification that is attached to an incoming message. We call it *ClientOrder*. The root of *ClientOrder* is *client*. So a client may send as many jobs as he likes and

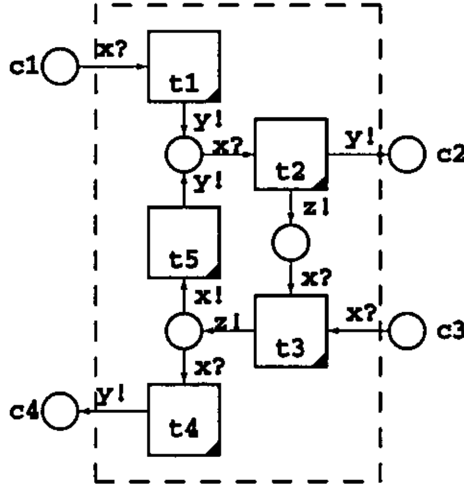

Figure 14.3: o-actor for Job

they will obtain their own identification internally. The m-complexes for c_2 have *job* as root. The complexes for connector c_3 consist of one simplex *job* that is also the root simplex. We call them *TaskOut* and *TaskIn* respectively. Finally the m-complex for c_4 has *client* as root. We call it *Product*. Note that all internal places have *ShopOrder* as complex class. It is assumed that the client gets the product symbolically in the form of the job identity. Note that *job* and *client* are k-simplexes, while all the others are v-simplexes.

The next step is the definition of appropriate value types for the complex classes. This is straightforward in this case:

- $ClientOrder := [\, c : ID,\ t : (RT \times DU)^*]$,
- $ShopOrder := [j : \ ID,\ c : \ ID,\ t : (RT \times DU)^*]$,
- $TaskOut := [j : \ ID,\ t : RT \times DU]$,
- $TaskIn := ID$,
- $Product := [c : ID,\ j : ID]$.

Here RT is a type that denotes the resource types and DU is the type for the durations of tasks. We may choose here $\mathbb{Q}$ or a restricted form of it that allows only non-negative values. Note that attribute t denotes the set of tasks. Since a task is identified by its successor task (except for the last one), we may use this representation. Now we are ready to specify the processor relations. They are straightforward as well.

t_1
$x? :\ ClientOrder$ $y! :\ ShopOrder$
$\pi_j(y!) = New \ \wedge\ \pi_c(y!) = \pi_c(x?) \ \wedge\ \pi_t(y!) = \pi_t(x?)$

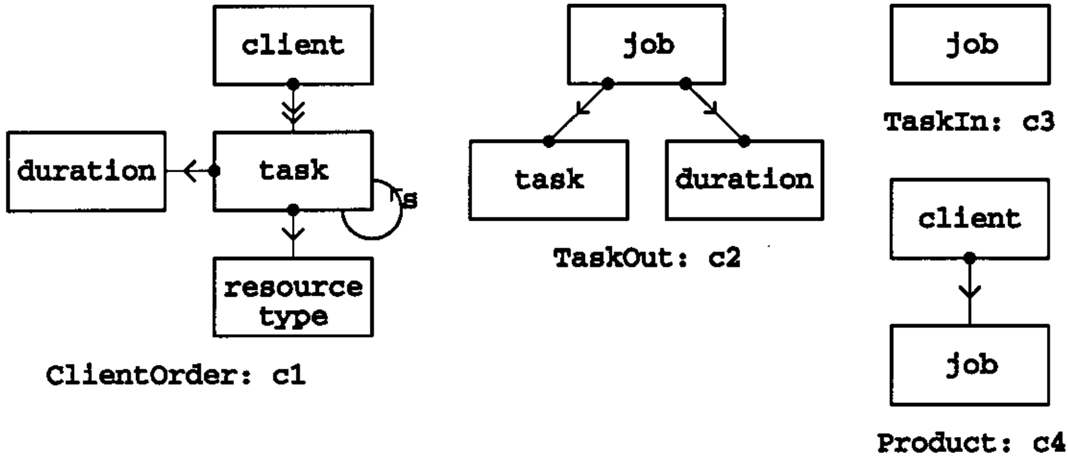

Figure 14.4: m-complex classes for Job

t_2
$x? : ShopOrder$ $y! : TaskOut$ $z! : ShopOrder$
$\pi_j(z!) = \pi_j(x?) \ \wedge \ \pi_c(z!) = \pi_c(x?) \ \wedge \ \pi_t(z!) = tail(\pi_t(x?))$ $\pi_j(y!) = \pi_j(x?) \ \wedge \ \pi_t(y!) = head(\pi_t(x?))$

t_3
$x? : ShopOrder$ $y? : TaskIn$ $z! : ShopOrder$
$y? = \pi_j(x?) \ \wedge \ z! = x?$

t_4
$x? : ShopOrder$ $y! : Product$
$\pi_t(x?) = \langle\rangle$ $\pi_c(y!) = \pi_c(x?) \ \wedge \ \pi_j(y!) = \pi_j(x?)$

t_5
$x? : ShopOrder$ $y! : ShopOrder$
$\pi_t(x?) \neq \langle\rangle$ $y! = x?$

Note that we only displayed the main schemas for these processors. The time does not play a role in this part of the system.

The next o-object we are considering is *Resource*. The o-complex class for *Resource* is displayed in figure 14.5. We call this class: *Machine*. It satisfies a tree constraint with *resource* as root. The simplex class *job* is needed to memorize which for which job the resource is working, if

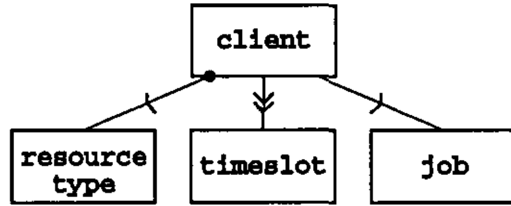

Figure 14.5: o-complex class for Resource: Machine

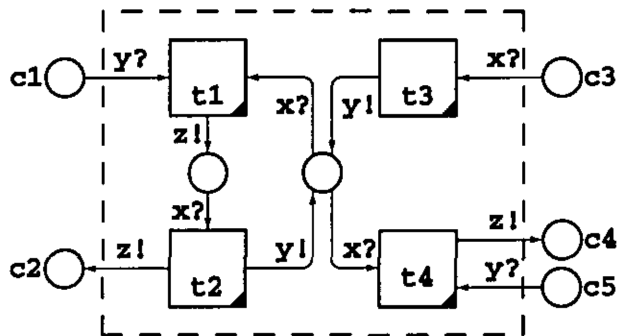

Figure 14.6: o-actor for Resource

it is not idle. The simplex class *timeSlots* represents free time slots for the resource to work for tasks. A time slot is a pair of rational numbers. The o-actor for *Resource* is displayed in figure 14.6. All processors are functional and complete. Only t_2 and t_3 are total. The others have to find a right resource, which is their precondition. Processor t_3 creates a new machine by sending a message that has the same format as the machine data. Processor t_4 deletes a machine with an identity given by *ResourceManagement*. (Note that *ResourceManagement* should be able to remember the machines it has created.)

The m-complex classes for the connectors c_1 and c_2 are *TaskOut* and *TaskIn* respectively. The m-complex classes for connectors c_3 and c_4 are the same as the o-complex class for *Resource*, because we assume that *ResourceManagement* puts and takes complete resources. The m-complex class for connector c_5 is trivial: only the identity of the resource is in the message. We call this class: *Retrieve*. The value types for the complex classes are:

- $Machine := [r : ID,\ k : RT,\ j : ID,\ s : \mathbb{F}(\mathbb{Q} \times \mathbb{Q})]$

- $Retrieve := ID$

Now we are ready to specify the four processors of *Resource*. They are very simple in this case. Processor t_1 selects a suitable resource for a task and determines the delay of it (by means of $z_t!$).

t_1
$x? : \mathit{Machine}$ $y? : \mathit{TaskOut}$ $z! : \mathit{Machine}$
$\pi_1(\pi_t(y?)) = \pi_k(x?)$ $\exists t : \mathbb{Q} \times \mathbb{Q} \bullet t \in \pi_s(x?) \wedge$ $\quad \pi_1(t) \leq \mathit{TransTime} \ \wedge \ \pi_2(t) \geq \mathit{TransTime} + \pi_2(\pi_t(y?))$ $z! = x? \oplus \{j \mapsto \pi_j(y?)\}$ $z_t! = \mathit{TransTime} + \pi_2(\pi_t(y?))$

t_2
$x? : \mathit{Machine}$ $y! : \mathit{Machine}$ $z! : \mathit{TaskIn}$
$y! = \oplus\{j \mapsto \perp\}$ $z! = \pi_j(x?)$

t_3
$x? : \mathit{Machine}$ $y! : \mathit{Machine}$
y!=x?

t_4
$x? : \mathit{Machine}$ $y! : \mathit{Retrieve}$ $z! : \mathit{Machine}$
$\pi_r(x?) = y?$ $z! = x?$

This example was very simple however it demonstrates the object oriented method well. Note that we have seen different models for almost the same real-world systems in which *jobs* are asking for *resources*. Sometimes we modeled it such that the resource is "carrying" the job (like we did here) and sometimes we defined a new object class *operation* (like we did in chapter 13). The last solution has the advantage that the choice to add the job to the resource instead of the resource to the job, is avoided. This example shows that there are many ways to model reality.

References and Further Reading

There is not much literature on methods for *making* a model; most literature is about *frameworks* for modeling and about *phasing* the modeling process (the development *life cycle*). The reason that there is so little theory on the modeling methods is that modeling is an *art* rather than a *science*. For actor modeling [Genrich and Lautenbach, 1981], [Peterson, 1981] and [Jensen, 1990] are good references. Further there are many modeling examples published, for example [Brauer, 1980].

A special modeling method based on Petri nets is found in [David and Alla, 1989]. The modeling of time aspects can be found in [van der Aalst, 1992] and the modeling of continuous processes in [David and Alla, 1990]. In [van der Aalst, 1992] a modeling approach for *logistic systems* is presented. Structured actor models (such as free choice nets) are in fact Petri nets and [Reisig, 1985] and [Peterson, 1980] are good references. The transformation to valueless models, often called *unfolding*, is due to K. Jensen, see [Jensen, 1992]. In the Springer-Verlag Series *Advances in Petri Nets 19XX* and in the *Proceedings of the XX-th International Conference on Applications and Theory of petri Nets* many applications are recorded.

For *object modeling* there is a method based on a slightly different binary data framework, called NIAM, see [Nijssen and Halpin, 1989]. Another approach is offered in [Rishe, 1988]. In most books on data models the transformation to the relational data model is considered. In [Spaccapietra, 1987; Teorey *et al.*, 1986] many aspects of modeling with the entity-relationship data model are studied. In [Brodie *et al.*, 1984] several different approaches of object modeling and in particular constraint specification are given. For the transformation of the (our version of) the functional data model to the relational data model see [Aerts *et al.*, 1992]. The transformation of the nested relational data model to the relational data model can be found in [Paredaens *et al.*, 1989].

Object oriented modeling is a popular topic. In [Sibertin-Blanc, 1991] an approach for object oriented modeling with Petri nets is given. In [Coad and Yourdon, 1990] and [Rumbaugh *et al.*, 1991] two approaches are offered based on informal frameworks, however many ideas can be translated to our frameworks. In [Sernadas *et al.*, 1991] and [van Assche *et al.*, 1991] several ideas for object oriented modeling of information systems are given. For database systems to be object oriented there is a set of requirements formulated in [Atkinson *et al.*, 1989]. Many ideas of object oriented programming can be applied to modeling, see [Meyer, 1988; Booch, 1991]. There is an object oriented method that has some similarity with ours, called HOOD, see [Di Giovanni and Iachini, 1990].

In [Jackson, 1983] a different but "complete" modeling method (for actor and object modeling) is given. In [Ward and Mellor, 1985] a modeling method for an (informal) framework based on data flow diagrams, the entity-relationship model and time is treated. In [Sol and van Hee, 1991] different modeling methods for complete systems are given, among

which the approach given in this book.

Exercises

1. Model a flip-flop as a classical Petri net.

2. Model a machine that can count objects that are coming from some generator in the p-ary number system up to n digits, as a classical Petri net.

3. Consider a teller machine, i.e. a machine with the following functions:

   - to get money a person has to put his card into the machine, and then he has to enter his personal code,
   - if his code is correct he may enter the amount he wants,
   - if his balance is larger or equal than the amount, he gets the money and the amount is subtracted from his account, otherwise he gets no money.

   Assume the machine has infinite capacity for money and that entering a wrong code or amount can not be corrected by the user.

   (a) Model this system as an actor model.

   (b) Extend the functionality by allowing people to put money in the machine, which will result in an update of the account. Answer (a) again.

   (c) Extend the machine by allowing people to transfer money to the account of somebody else. Answer (a) again.

4. Consider a simple railroad system with one track that consists of a closed curve without intersections. The track is divided into 5 sections, each ending with a semaphore that is either red or green. Each semaphore has a sensor that tells if a train has (completely) passed the semaphore. There are two trains riding in the same direction. The information system has to guarantee that:

   - no two trains are allowed to be in the same section,
   - if the section after a semaphore is empty the semaphore should be green,
   - there is no deadlock.

   Assume these requirements hold in the starting state.

   (a) Model the railroad system including its information system as a classical Petri net. Explain what the objects, places and processes represent in reality.

   (b) Modify the system in the sense that the track intersects with itself and add the requirement that collisions should be excluded.

5. Consider a medical care system in which ill persons see a family physician first. The family physician can take one of the following decisions:

   - he can give the patient a medicine and after a while he wants to see the patient again,
   - he can decide that the patient cannot be treated (then the patient leaves the system),
   - he sends the patient to a consulting physician.

   The consulting physician can make the first two decisions the family physician can make, but in addition he can do some further medical examination: a blood test or X-ray photographs or both. He only wants to see the patient back, if all examinations have been done. An extra decision he can make is that he can send the patient back to the family physician. The physicians base their decisions on the number of visits, the used medicines and the blood tests and X-rays of the patients.

   a Make an (incomplete) actor model for this system.

   b Modify the model in such a way that there is an arbitrary number (of both types) of physicians and that each patient has to be seen by the same physician at each successive visit.

6. The Car Rental Company (CRC) has many stations in the country, where they store and maintain cars. Customers make a reservation for a type of car at some station for a specific period of time. When the customer arrives at the station on the first day of the rental period, a car of the right type is assigned to the client. The client may also cancel a reservation, however before the rental period starts.
   A client may return a car to another station at the end of his rental period (this will be charged). If a client wants to extend his rental period, this will be considered as a new rental.
   Cars can be in service, rented or shipped from one station to another (by CRC).
   The information system must be able to keep track of the cars and the reservations and it must support the process of car assignment and invoicing.

   (a) Make an object model for CRC, including graphical constraints and (if necessary) additional constraints in natural language and predicate calculus.

   (b) Make an (incomplete) actor model.

7. Make an actor model for the following Resource Reservation System (RRS). The system receives requests from clients for an arbitrary resource on a particular date. (Resources are for instance

seats in a concert hall or in an airplane.) The client receives an acknowledgement of his request. If there is a resource free for that particular date, a reservation is made and the client receives a confirmation telling the number of the resource. If no resource is available the request will wait until somebody else cancels his reservation. Clients may cancel their requests or their reservations. The systems administrator should have facilities to delete all the reservations and requests if the date has expired. Consider two cases: one where the unsatisfied requests are assigned to a resource in an arbitrary way and one in which they are served in a first-come-first-served order.

8. Make an object model for the Resource Reservation System of the former exercise that can be used to define the database of a monitoring information system for the system (i.e. the universal complex class belongs to the store of the monitoring information system).

9. Make a (complete) actor model in the object oriented style of a university. Consider the following o-classes: student, instructor and course. Choose appropriate o-complex classes and life cycles.

10. Modify the actor model of the railroad station (see text) such that the station master will be able to decide to which track a new train will go.

11. Modify the construction for token cancellation (see text) such that actor Z can select tokens to be cancelled.

12. Make an object model for the store of the train control system displayed in figure 12.8.

Index

! decoration, 49, 73
′ decoration, 49, 73
(;), 85
=, 65
? decoration, 49, 73
A^n, A^*, A^∞, A^+, 85
X-similar $\sim_X$, 89
$\Pi_{\ldots}$, 66
$\perp$, 65
$\cdot$, 69
$\cup$, 66
ϵ, 24, 85
$\oplus$, 67
π_ℓ, 66
σ, 88
τ, 88
k-bounded net, 167

A, 104
absolute time, 272
abstract simplex, 177
Ackermann function, 346
active domains, 172
active objects, 217
activity network, 166
actor, 15, 47
actor framework, 84
actor model, 107
actor model properties, 112
actor modeling steps, 137
actor roles, 147
aggregate, 194
antithetic variates technique, 290
applicable firing assignment, 110
applicative order reduction, 321
association simplex class, 178
attribute domain, 203
attribute simplex class, 178
automated systems, 21
autonomous behavior, 25, 86
autonomous trace, 86

base, 253
basic type, 63, 301
bisimilar, 89
bounded nets, 167
bounded occurrence, 317
breadth-first search, 264
broadcasting, 152
business systems, 16

C, 103, 104
CA, 107
cancellation token, 158
canonical form, 247
cardinality constraint, 41, 97
cat, 67
CB, 94
channel, 47, 122
class diagram, 37
class model, 93
classical Petri nets, 47, 163
client-server, 219
closed actor, 54, 104, 106
CM, 101
CN, 93
com, 95
complex class, 35
components, 302
composition of actor models, 119
composition of object models, 118
compound object, 177
concrete simplex, 177
conflict free, 164, 279
congruential method, 287
connector, 47
constants, 301
constraint, 96
constraints, 28, 40
construction model, 11

consumption function, 237
cont, 100
context actor, 137, 139
continuous processes, 159
control variates technique, 289
CR, 94
critical path method, 279
CT, 107

$D_{r,c}$, 96
data oriented, 134
dead set, 242
deadlock, 25, 87, 242
decomposition guidelines, 140, 145
defined predicate, 72
delay, 272, 277
depth-first search, 264
deterministic transition law, 26
deterministic transition system, 87
direct addressing, 152
discrete dynamic systems, 12
DK, 97
DM, 94
domain class, 39
domain exclusion constraint, 97
domain key constraint, 43, 97, 353
domain type, 310
DX, 97
dynamic programming, 345

E, 86
eager autonomous behavior, 27, 88
earliest arrival time, 278
empty row, 302
empty sequence, 302
empty set, 302
empty tuple, 302
entity simplex class, 178
entity-relationship schema, 208
environment, 161
evaluation function, 315, 321
event, 24, 86
exclusion constraint, 43, 97, 354
executable specifications, 297, 298
expressive comfort, 297
expressive power, 297
extendible language, 298
external events, 27, 118

F, 107, 108
f, 110
FA, 110
factorial function, 349
fairness, 155
FC, 97
file as one token, 149
file as set of tokens, 151
filter, 236
finite mathematical value, 298
finite state machine, 165
firing assignment, 110
firing rules, 108
firing variable, 248
flat net model, 103
flow balance, 238
flow function, 237
flow matrix, 235, 238
FN, 107
formalism, 13, 83
framework, 83
free choice nets, 164
free constraint, 101
free value universe, 303
free variable occurrence, 318
function application, 68
function declaration syntax, 331
function definition syntax, 323
function graph, 325
function signature, 310
function universe, 309
functional dependencies, 204
functional equation, 249
functional model, 11
functional object model, 201
functionality, 43, 51, 97, 122
functions, 65

global constraint, 46, 101, 173
graph of function, 90
graphical representation, 52
guidelines, 133

HA, 104
head, 67

hierarchical net model, 104
history, 192
HL, 105

I, 103, 104
i subscript, 73
IC, 100
ID, 107
identity filter, 237
if then else fi, 66
IM, 101
independent place invariants, 244
induced transition system, 111
information preservation, 200
information simplex, 177
information system, 18, 137
inheritance constraint, 43, 100, 354
initial event, 24
injectivity, 43, 97
input completeness, 51, 121
ins, 67
instance, 38, 94, 95, 199, 204
intelligent information systems, 20
inter-organizational information systems, 20
interval-timed actor model, 271
invariance properties, 61
invariant place property, 240
inverse transformation method, 287
irreducible data model, 203
isomorph, 91, 209
iterated application, 342

join ⋈, 64

key constraint, 43, 97
knowledge, 217, 218

$\mathcal{L}$, 86
L, 103, 104
$\ell(p)$, 87
lambda calculus, 298
latest arrival time, 278
lazy function, 311
lexicographical ordering, 306
life cycle, 28, 217
limit of a monotonous sequence, 341
linear recursive functions, 342
live processor, 241
livelock, 25, 89, 117, 280
local constraint, 173

M, 104, 105
m-complex class, 218
map construction, 69
map term, 316
marking, 236
maximal autonomous behavior, 86
maximal autonomous trace, 86
maximal exclusion constraints, 180
measurement actors, 139
memoryless transition system, 87
message, 217
meta syntax, 313
method, 131, 217
method of successive approximations, 342
minimal key constraint, 180
minimal support invariant, 254
model, 83, 199
model making, 61, 131
model transformation, 131
modeling language, 298
molecular object, 177
monitoring information systems, 19, 197
monomorphic function, 65, 310
monotonous function, 341
monotonous sequence, 341
monotonous transition system, 88, 114
multi-valued dependencies, 204
mutual exclusion, 155

$\mathcal{N}$, 85
negative correlation, 289
nested relational schema, 212
New, 60, 73
Newton-Raphson method, 346
non-deterministic transition law, 26
non-elementary actor, 47

non-negative place invariant, 242
non-strict function, 311
normal form, 203
normal form of a set, 307

O, 103, 104
o-actor, 218
o-complex class, 218
o-object method, 219
object, 15
object framework, 84, 199
object life cycle, 154
object model, 35, 101, 199
object oriented, 134
object oriented frameworks, 200
object oriented modeling, 217
object roles, 146
object universe, 96
occurrence graph, 263
office information systems, 20
OM, 107
open actor, 53, 104, 106
OU, 96
output completeness, 51, 121
overloading, 68, 309

P, 103, 104
pair, 302
parent function, 107
partial functions, 68
path, 24, 89
PC, 101
Petri filter, 237
pick, 67
place invariant, 144, 234, 238
planning, 192
polling, 168
polymorphic function, 65, 310
positive place invariant, 243
predicate, 71
predicate syntax, 330
prefix p^i, 85
prefix of a trace, 24
prefix-closed, 85
primary key, 203, 208
primitive recursive function construction, 344
process oriented, 133
processing time, 156
processor, 47
processor characteristics, 51, 121
processor execution rules, 54
processor relation, 49, 108
product type, 304
product type constructor, 304
production function, 237
protocol, 219, 263
prototype, 283

QΦcoverability tree, 264

R_p, 108, 109
$R_{r,c}$, 96
range class, 39
range exclusion constraint, 97
range key constraint, 43, 97
range type, 310
reachability, 263
reachable states, 25
realizable, 261
recursion, 322
recursion operator, 340
recursive functions, 70
referential integrity, 204
regression analysis, 286
regular values, 308
relation, 204
relational data model, 200
relational data modl, 203
relational instance, 203
relational schema, 203
relationship constraint, 97
relationship path, 98
relative time, 272
representation function, 96
rest, 67
RG, 94
RK, 97
RN, 93
root simplex class, 101
root simplex class, 101
row, 64
row constructor, 302
RX, 97

safe net, 167
SC, 101

schema, 50, 72, 73, 199
schema definition semantics, 336
schema definition syntax, 334
schema equality, 336
schema expression syntax, 333, 334
schema operator, 74
schema universe, 333
scope, 317
script, 75, 337
sequence, 64, 85, 302
sequence constructor, 302, 304
sequential process, 153
serializability, 116
set, 64
set constructor, 302
set restriction, 68
set theory, 298
set type, 304
signature, 66, 323
sim, 94
similarity, 89, 274, 275
simple singular value, 308
simplex class, 34
simulation, 233
singular value, 308
SN, 93
specification, 61
specification language, 298
St, 109
stable function, 341
stage, 219
standard constraint, 97
standard term, 318
standardizing, 318
state, 23, 39, 109, 219
state machine, 153, 165, 218
state space, 23, 88, 109
static type system, 297
store, 47, 106, 122
strict function, 311
strongly memoryless transition law, 28, 88
successive approximations, 341
suffix-closed, 85
support, 254
surjectivity, 43, 97
synchronization, 154
syntactical transformation function δ, 335
syntax base, 313
system composition, 30, 118

T, 88, 107
t subscript, 73
tail, 67
target system, 137
TC, 100
terms, 69, 317
time, 111
time dependent, 192
time domain, 24, 88, 107
time-out, 157
timed colored Petri nets, 49
timeless actor models, 163
$T\ell$, 87
token, 15, 35, 109
token identification, 55
token priority, 159
token time, 54, 145
top, 104
totality, 43, 51, 97, 121
tr, 110
trace, 24, 86
transaction, 219
transition balance, 260
transition invariant, 234, 260
transition law, 25, 86, 111
transition relation, 87, 110
transition system, 86
transition systems framework, 84
transition time, 55, 111
TransTime, 60, 73
trap, 242
tree constraint, 45, 100, 353
tuple, 64, 204, 302
tuple compatibility, 307
tuple equivalence, 307
tuple join, 307
tuple type constructor, 304
type, 39
type and value constructors, 64
type checking, 233
type definition syntax, 314
type definitions, 68
type function, 318

type universe, 304
type variable, 65
typed lambda expression, 316
typed set theory, 298

universal complex class, 37, 93
universal constraint, 173

validation, 132
value, 39
value simplexes, 218
value universe, 305
valueless actor models, 162
verification, 61, 132

W_p, 89
weights, 238
well-typed function declaration,
332
well-typed function definition, 324
well-typed predicates, 330
well-typed schema, 334

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.

91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.

92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.