Statecharts as Protocols for Objects

Annette Laue
Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany
3laue@informatik.uni-hamburg.de

Matthias Liedtke
Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany
3liedtke@informatik.uni-hamburg.de

Daniel Moldt
Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany
moldt@informatik.uni-hamburg.de

Ivana Tričković
Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany
trickovi@informatik.uni-hamburg.de

Abstract

For the specification of object-oriented systems, usually several models representing different views are developed. The necessary integration of views is often delayed until implementation, but for validation and verification purposes the merging of views is desirable already during specification. Motivated by this, a model of objects is proposed in which the different views on an object are regarded as independent and well-connected entities that are encapsulated inside of it.

Reference nets, a powerful high-level Petri net formalism, are used as an underlying uniform modeling technique for objects and the two views on them considered here — protocols and functional parts. Protocols are defined as statecharts and mapped into Petri nets. Models become executable by utilizing Renew, a tool for drawing and simulating reference nets. In an example, the main ideas of the suggested approach are clarified.

1 Introduction

The development of approaches for the specification of systems is a major task in computer science. Today, most suggestions in this area are based on the paradigm of object-orientation. To describe object-oriented systems, usually several models representing different views are developed. Generally, every model is illustrated using a certain *technique*, which seems to be the most appropriate one in a given context ([24]). Graphic techniques (diagrams) are a usual choice for this. With regard to the area of graphic modeling languages for object-oriented systems, UML ([25]) is a de-facto-standard.

The price that has to be paid for a clearer modeling on the basis of views is the necessary integration into one overall model. Concerning implementation, this effort can be supported by a tool. For every class, such a tool may merge the relevant aspects of different views by generating code bodies or even executable code out of the constructed diagrams. But in order to improve possibilities for validation and verification processes, a clear separation of views at the level of classes together with well defined connections between them should be established in specification. It is desirable that such a form of specification eases changes concerning individual views.

In this context, the concept of Abstract Data Types ([19]) is of particular interest. The definition of an Abstract Data Type (ADT) encompasses the allowable operation names, their signatures, and a set of axioms which determine the meanings of operations ([9]). Furthermore, the axioms characterize the appropriate handling of the ADT; one can say that they also describe the *protocol* of the ADT.

Motivated by this, we propose that every class consists of two parts. One of these parts explicitly represents the protocol of the class whereas the other one covers its functionality. The protocol part of a class describes its dynamic behavior which is related to control aspects. It is modeled as a statechart. The functional part contains everything which does not concern control aspects, e. g., data transformations and attributes. Contrary to UML, a statechart is not just a view on a class, but an inherent part of it. All messages sent to an object are handled by its statechart. It

becomes possible to interpret the protocol of an object as a *first class citizen* that can be handed over to other objects. The separation of protocol and functionality is a first step on the way to a form of specification which permits the integration of different views at the level of classes.

With regard to distributed systems, an unchangeable protocol is not always reasonable. For example, an object can migrate, and its new environment may require a new handling. Therefore it should be possible to allow a straightforward adaptation of object behavior. Due to the clear separation suggested here, a change of environment can be taken into account easily by enabling a change of behavior via exchange of protocol. Of course, this requires some knowledge about the cohesions of the protocol and the functionality.

In the following section, the basic concepts of statecharts are presented. Due to the use of reference nets, a high-level Petri net formalism, as the underlying uniform modeling technique the main ideas of mapping statecharts into Petri nets are explained, too. Additionally, section 2 deals with the most important notions of reference nets. In section 3, the separation of protocol and functionality is covered in more detail. An example showing a possible application of the suggested approach is described in section 4. Section 5 concludes the considerations presented here and provides an outlook.

2 Statecharts and Reference Nets: Basic Notions

In this section, statecharts and reference nets are presented. Section 2.1 deals with statecharts, which are chosen for the representation of protocols. In section 2.2, reference nets are introduced. These nets are used in sections 3 and 4 to illustrate the proposed separation of protocol and functionality of an object.

2.1 Statecharts

In 1983, David Harel developed the visual formalism of statecharts, a powerful extension of conventional finite automata. Statecharts allow a compact representation of complex behavior of reactive systems. The type of approach which is chosen for system decomposition determines the kind of entities for which one wants to illustrate dynamic aspects using statecharts.

In the beginning, Harel has presented statecharts as a sort of diagram for a system-wide state space that is detached from such an approach ([10]). In that context one single statechart describes how a complete system responds to certain events by changing state and raising actions. However, in case of complex systems, the embedding of statecharts into an overall approach is inevitable. Thus, statecharts early became part of an approach influenced by Structured Analysis ([7, 14]).

The emergence of object-oriented approaches has led to proposals how to use statecharts for the description of object behavior (e. g., see [27, 6, 11]). Today, statecharts are the most popular technique for illustrating life-cycles of objects, and they constitute an essential part of UML ([25]).

In section 2.1.1, the advantages of statecharts over conventional finite automata are summarized briefly, and the choice of statecharts for the representation of protocols is justified through this. Then the most important elements and features of statecharts are described in section 2.1.2. This presentment is based on [10] and [25]. In section 2.1.3, the question of an adequate semantics is considered. There the mapping of statecharts into Petri Nets is suggested in order to obtain a precise semantics. In section 2.1.4, the essential aspects of a corresponding proposal ([24, pp. 261–338]) are explained.

2.1.1 Advantages of Statecharts

State diagrams of conventional finite automata show some serious drawbacks ([20, 13]) that can be overcome by using statecharts ([10]). In this context, there are two concepts of particular significance: the development of a *hierarchy of states* (a *state tree*) and *orthogonality*. A state tree permits top-down- and bottom-up-methods during system specification and leads to a more forseeable representation because several state transitions can be subsumed. The concept of orthogonality encompasses the separation of a state into two simultaneously active components whereby the exponential explosion of state space is avoided and concurrency becomes possible.

2.1.2 Important Elements and Features of Statecharts

The state tree of a statechart is illustrated by graphic inclusion of states. If an entity is in a state decomposed according to *XOR decomposition* (or-state), exactly one of its substates is active. In case of *AND decomposition* of a given state (and-state), all of its *components* are active at the same time. Components are visually separated by dashed lines and always represent or-states. *Basic states* do not have substates. At every layer of abstraction, at most one state in the set of substates of an or-state may be a *default state*. Additionally, object-oriented statecharts can have a *final state* that indicates termination. It is also possible that a statechart encompasses several final states which stand for the ending of activity at certain levels.

State transitions are represented by arrows that may be inscribed with a triggering *event*, a *guard*, and an *action* which is raised during the actual change of state; all of these inscriptions are optional. Due to different kinds of events and actions several variants of communication can be described. Furthermore, an action can be bound to a state. In this case it is carried out every time a state is entered or left. It is also possible that a state is connected with an activity which is carried out throughout the whole time the state is active.

Statecharts can also include *submachine states*, *synch states*, *stub states*, and *pseudo states*. Submachine states enable the integration of already existing statecharts by providing a shorthand notation for them. Synch states allow for synchronization of orthonogal components. Stub states are used inside of submachine states and represent states of referenced submachines. Among pseudo states, one can distinguish between the aforementioned default vertices, two kinds of *history vertices*, *fork* and *join vertices*, as well as *junction* and *choice vertices*. By history vertices the most recent state configuration is restored if a state is reentered. There are two types of history vertices: *deep history* and *shallow history*. Fork and join vertices are used in conjunction with and-states. By them, transitions can be split and merged. Junction and choice vertices realize static and conditional branches, respectively.

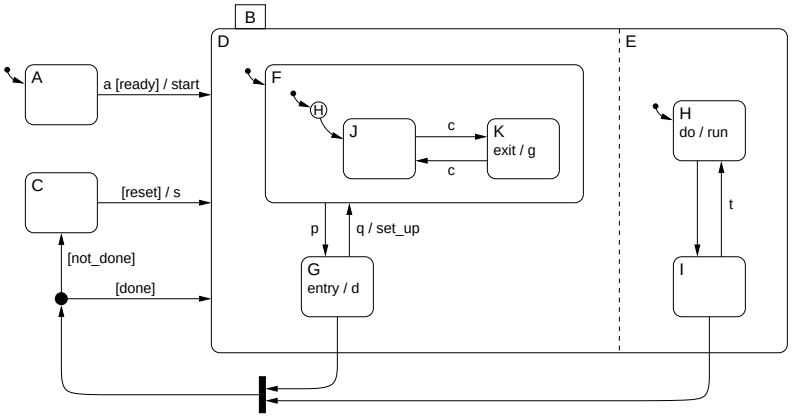


Fig. 1. An example of a statechart.

Figure 1 shows an example. Beside other constructs, an and-state, a shallow history vertex, a join vertex, and a junction vertex are used. More detailed descriptions of elements and features of statecharts can be found in [10] and [25].

2.1.3 Semantics of Statecharts

If the process of modeling is supported by the use of statecharts, one has to fix their semantics. In early days, semantics of statecharts has already been considered. Firstly, one question was of significant importance: Can effects of an actual change of state initiate further state changes inside of the step under consideration? This question has led to different suggestions for an appropriate semantics of statecharts (e. g., see [26, 12]). However, because statecharts are used for several domains and purposes, numerous variations and semantics have been proposed since their emergence; [3] provides an overview.

With regard to object-oriented statecharts, semantics has been considered barely at first ([27]). In contrast to this, the metamodel of UML explicitly distinguishes between abstract syntax, static semantics, and dynamic semantics ([25]). But the definition of statecharts by the UML metamodel shows two essential disadvantages: dynamic semantics is described in terms of natural language, and for the three aspects three different means of description are used (class diagrams, object constraint language, and natural language, respectively).

If statecharts are mapped into Petri nets, all aspects can be described with one single graphic technique. By using Petri nets (or simply: nets) an operational semantics for statecharts in terms of a graphic representation is obtained. Furthermore, the visualization as a net allows using the "nets in nets" paradigm to illustrate the approach suggested here: the protocol statechart of an object can be represented by a net; the object itself is another net where this protocol can be accessed via a token that denotes a reference to the protocol net (see section 3.3). In this way, it is possible to provide a formal basis for the idea of dividing an object into a protocol and a functional part.

2.1.4 Mapping Statecharts into Colored Petri Nets

In this section, just the essential ideas behind a mapping of statecharts into colored Petri nets are represented. For a deeper understanding, [24] is an appropriate source, which is also the basis for the following. A good introduction for colored Petri nets is [15]. With regard to [15], the mapping of statecharts into colored Petri nets requires another concept of refinement: A place is not *replaced* by its refinement, but it is *extended* by its refinement.

Basic states and or-states are replaced by places. According to statecharts, substates are arranged inside the place representing the corresponding or-state. A so-called *validity token* residing in this place is used to show that the or-state is active. Arrows connecting states of the given statechart are replaced by transitions. Each time an or-state becomes active or inactive, the involved transitions have to ensure that one of its substates becomes also active or inactive simultaneously. Every transition representing an arrow has to be connected to another place, where the triggering event resides as a token.

For a default state, a so-called *default event* and an additional *default transition* are specified: the transition leading to the or-state deposits the *default token* at the place of the or-state, and the default transition deposits *validity tokens* at the right places. The concept of history can be transfered, if only the token for the or-state is removed in case of leaving the or-state. One can also provide a certain *history event*.

An and-state is represented by a place inside of which its components are arranged as places drawn with a dashed border. Entering an and-state leads to the depositing of a validity token at the corresponding place. Furthermore, a default token is put onto every place representing a component. Inside of components, default tokens will be transformed into validity tokens.

The explained representation of a statechart as a net is not as compact and clear as the original one by Harel, of course. Nevertheless, it offers the advantages mentioned in section 2.1.3. Furthermore, by using reference nets, which will be described in the following, simpler net structures for statechart constructs than the ones described in [24] seem to be possible. The questions coming from this topic will be a subject of future research. The statechart representations shown in sections 3.3 and 4 are limited to very simple protocol nets because the interplay of protocol and functionality is of major interest there.

2.2 Reference Nets

Reference nets are a high-level Petri net formalism that incorporates the "nets in nets" paradigm ([30]). They extend the basic concepts of colored Petri nets ([15]) by dynamic creation of net instances, references to other net instances as tokens, and dynamic transition synchronization via synchronous channels ([17]). Their inscription language is based on Java.

In section 2.2.1, the basic concepts of reference nets are introduced. References to net instances and synchronous channels are explained in sections 2.2.2 and 2.2.3, respectively. Section 2.2.4 provides some aspects of the Renew tool, which was used for constructing the nets shown in sections 3 and 4. For a deeper understanding of reference nets and the Renew tool, [18] should be considered.

2.2.1 Net Elements, Tokens, and Basic Features

Like all variations of Petri nets, reference nets consist of *places* (graphically represented by circles), *transitions* (rectangular boxes), and *arcs*. Besides ordinary *input* and *output arcs*, that come with exactly one arrow head and remove or deposit tokens at a place the usual way, there are *reserve arcs* (exactly two arrow heads) and *test arcs* (no arrow heads at all). Whereas reserve arcs are just a shorthand notation for one input and one output arc, test arcs behave differently. They access tokens in input places without reserving them during the firing of a transition. Therefore, a single token may be accessed by several test arcs at once.

Each place or transition can have a *name*, displayed in bold type. Every net element may be assigned semantic inscriptions. Places can have a *type* and an arbitrary number of *initialization expressions*. If a net instance is created (see section 2.2.2), the initialization expressions are evaluated and the results represent the initial marking of the net. By default, a place is initially unmarked. An arc may have an optional arc inscription that is evaluated in case of the firing of a transition. According to the result, tokens are consumed and created.

Transitions can carry several kinds of inscriptions: *expression inscriptions*, *guard inscriptions*, *action inscriptions*, *creation inscriptions*, and *synchronous channels*. Expression inscriptions are Java inscriptions which are evaluated while the net simulator searches for a binding of the corresponding transition. The result of this evaluation is discarded, but in such expressions one can use the equality operator `=` to influence the binding of variables that are used elsewhere. Guard inscriptions are preconditions to transitions, i. e., a transition is activated only if all attached guard conditions evaluate to `true`. They are expressions that are prefixed with the reserved word `guard`. Action inscriptions start with the keyword `action`. They can help to calculate output tokens, and they are required for expressions with side effects. Contrary to expression inscriptions, action inscriptions are guaranteed to be evaluated exactly once during the firing of a transition. Creation inscriptions will be described in section 2.2.2, and synchronous channels will be dealt with in section 2.2.3.

There can be two kinds of tokens: valued tokens and tokens which correspond to a reference to some object. A valued token can be of any primitive data type offered by Java. Furthermore, the inscription language of reference nets has been extended to include *tuples*, so that valued tokens can also be tuples. E. g., `[7,7.0,"seven"]` denotes a 3-tuple which has as its components the integer `7`, the double precision float `7.0`, and the string `"seven"`. Tuples are useful for storing a whole group of related values inside a single token and hence in a single place. The inscription `[]` stands for a *black token*. By default, an arc transports such a token, but if one adds an arc inscription, that inscription will be evaluated and the result will determine which kind of token is moved. Among the tokens that denote references to objects, the most interesting ones represent references to net instances (see section 2.2.2).

During the firing of a transition, variables are bound to one single value. However, when the same transition fires the next time, the variables may be bound to completely different values.

2.2.2 Net Instances and Net References

In the notion of *object nets* ([31]), tokens of a so-called *system net* correspond to marked Petri nets on a lower level, called object nets. Since the object nets actually reside in the system net, we use the term *value-semantics* to describe this. Reference nets use another approach: The object nets do not actually reside in the system net, but tokens are references to object nets. Accordingly, multiple tokens can reference the same object net. In analogy to programming languages, we call this the *reference-semantics* approach.

When a reference net is constructed, it is specified as a static structure that serves as a template. During a simulation, it is used to create an arbitrary number of *net instances*. These instances have got a marking that can change over time. Whenever a simulation is started, a new net instance is created. Every net has to be given a *name*. New net instances are created by transitions that carry *creation inscriptions*. These consist of a variable name, a colon, the reserved word `new`, and the name of the net. E. g., the creation inscription `x: new net1` means that a new instance of the template `net1` is created and bound to the variable `x`. For any further access to new net instances their references, represented by tokens, are used. Each net instance comes with its own marking and can fire independently of other instances. A net does not disappear simply because it is no longer referenced. But, if a net instance is no longer referenced and none of its transition instances can possibly become enabled, then it is subject to garbage collection.

2.2.3 Synchronous Channels

Net instances need some means of communication. Reference nets use *synchronous channels* for this purpose, which were first considered for colored Petri nets by Christensen and Damgaard Hansen in [5]. They synchronize two transitions which both fire atomically at the same time. Both transitions must agree on the name of the channel and on a set of parameters before they can synchronize. Reference nets generalize this concept by allowing transitions in different net instances to synchronize. According to object-oriented programming languages, it is required that the initiator of a synchronization holds a reference to the other net instance.

The initiating transition must have a special inscription, a so-called *downlink*, which directs a request to a designated subordinate net. This inscription consists of an expression that must evaluate to a net reference (usually the name of a variable), a colon, the name of the channel, and an optional list of arguments. On the other side, the requested transition must be inscribed with a so-called *uplink*, which serves requests for every other net instance. Therefore, its inscription does not encompass an expression that designates the initiating net instance. Every time a synchronous channel is invoked, the channel expressions on both sides are evaluated and unified. In reference nets, `this` denotes the net instance in which a transition fires. Generally, transitions with an uplink cannot fire without being requested explicitly by another transition with a matching downlink. A transition can have an arbitrary number of downlinks, but at most one uplink. It is also allowed that a transition has both an uplink and downlinks.

Although there is a direction of invocation, this direction does not coincide with the direction of information transfer via the provided arguments. Indeed it is possible that a single synchronization transfers information in both directions.

2.2.4 The Renew tool

The models presented in sections 3 and 4 were constructed by using Renew (Reference Net Workshop), a tool for specifying and executing reference nets. Renew itself is a Petri-net-based software package developed by members of the Computer Science department of the University of Hamburg. It is implemented in Java and is freely available (URL: `http://www.renew.de`). It offers a GUI for building net models and viewing simulation runs. To be able to use reference nets and the Renew tool to their full capacity, some knowledge of Java is required.

3 Protocol and Functionality as Object Parts

As mentioned in the introduction, an Abstract Data Type (ADT) consists of operation names and signatures, but also of a set of axioms which constitute the meaning of the operations and thus determine the protocol of the ADT. Classes can be regarded as implementations of ADTs. However, by the elements of most object-oriented programming languages, only attributes and operations can be expressed. It is not possible to formulate separate pre- or postconditions. Another drawback of object-oriented implementations is that in specification a system is usually described using different views, but the views are not distinguishable in the implemented classes anymore. The model introduced in this section is a step towards overcoming these problems. In this model, an object consists of several parts which represent the different views on the object. It is not necessary to consider other entities in a system like components or subsystems, for example. Following the paradigm “everything is an object” ([16]), we assume that there exist only objects in a system (the system itself included). Thereby, our model is scalable, and it will be of interest especially for the specification of complex objects.

In this section, we will first look at some general aspects of communication between objects (section 3.1). Section 3.2 introduces our model of objects, and the last part (section 3.3) illustrates how the considered object parts protocol and functionality cooperate.

3.1 Communication between Objects

There are two general possibilities to describe the communication between objects (cf. [4]): on the one hand the exchange of *messages* between objects can be emphasized, on the other hand the interfaces of the individual objects can be the basis for description. In the latter case, the resulting specification describes the communication in terms

of *operations*[1]. The connection is as follows. An operation invocation is a message sent from the calling object to the called object. The receipt of such a message can be denoted as an *event* and causes reactions in the called object, e. g., the execution of the corresponding operation. Especially in case of asynchronous communication the distinction between message and operation is important, because the execution of the operation can take place much later than the sending of the initiating message. In general, three kinds of communication can be distinguished:

- In an *asynchronous communication*, the sender proceeds with his work immediately after sending the message. It does not wait for an answer from the receiver of the message.
- A communication is *synchronous* if the sender of the message suspends its work until it gets an answer from the receiver of the message.
- Finally, by a *rendezvous* we understand a communication in which the sending of a message, its receipt, the corresponding reaction, and the sending and receiving of the answer message coincide to one atomic unit.

In principle, the operations of an object cannot be invoked at every time with reasonable result, i. e., the set of operations that can be executed is restricted by the actual state of the object. This leads to the question how an object has to be used or which sequences of operation invocations are allowed in a given state. So there exists a certain (implicit or explicitly given) *protocol* the clients of that object have to follow (see [8, p. 361]). Another meaning of the term "protocol" refers to the interaction of objects as a whole. In this context, a protocol describes a fixed sequence of messages sent by the objects involved in the interaction. It is common in the area of distributed systems (cf. [29, pp. 17, 219–225]). This kind of protocol will be called *communication protocol*.

Both kinds of protocols are relevant for the interaction of objects. While the focus of a communication protocol is the principle structure of a *complete process* involving several objects, the former kind of protocol lays emphasis on the possible operation invocations on *one object*. The explicit representation of such a protocol is of interest in particular against the background of the so-called *Design by Contract* ([23]), which is derived from the concept of Abstract Data Types. Contracts are the central part in this model and represent the rules for invoking operations. They encompass pre- and postconditions and maybe invariances and are deposited at the object providing the services. Potential clients of this object have to check the validity of the preconditions before they invoke an operation. The server guarantees that the postconditions of an operation hold if the preconditions were valid before the invocation of the operation. In todays object-oriented programming languages, a separate definition of pre- and postconditions is not possible except for Eiffel (see [22]). As a result of the mixing of protocol and actual functionality in the operations, the preconditions are not transparent for the clients.[2] Moreover, it is impossible to reject an operation invocation which is not valid before the actual execution starts.

It is advantageous to have an explicit state based representation of the protocol, e. g., a statechart, which comprises the reactions to incoming events and is presented at the interface of the object. So the clients cannot only get information about the generally available operations, but also about restrictions for their invocation and effects of their execution. In addition, it is apparent for the client if it will get an answer only in case of operation execution or also if the invoked operation cannot be executed; this information may be important for the decision on which kind of communication should be used. If, in addition, it is possible to inquire for the current state, a client can find out which operations are *in fact* executable. In this case it is important to lock the server so that no other client can change the state before the operation is executed. The synchronization between two objects is a further step: the client makes the server enter a specific state, and the subsequent process of interaction between the objects is prescribed, but other clients cannot influence.

3.2 Objects Encapsulating Views

Normally, systems are specified on the basis of different views in order to reduce the complexity of the description by using several models. In object-oriented approaches, objects represent the constituting elements of a system, and

[1] The expression "method" is often used instead of the term "operation".

[2] Providing additional operations by which it is testable if an offered operation can be invoked does not seem practical, either.

the views — in earlier steps analyzed in separation — are integrated and realized inside of them. This is also true for views which concern more than one object, e. g., a process view, because they have to be permitted by the objects.

During implementation, the structuring based on several views on the system gets lost and a mixing of the information kept in the different models happens. As classes are the entities for structuring object-oriented systems, it is obvious to choose views which relate to objects or classes as far as possible. In this way, the transition from specification to implementation of the system is eased.

Another improvement of the specification can be achieved if the views on an object are modeled strictly separated from each other. In combination with clear relationships between the views this encourages the understanding of an object. Understanding is not only important during development, but also for modifications at a later point in time. Moreover, the strict separation eases modifications of aspects of a view. For views and their relationships, the principle of maximum cohesion (inside one view) and minimum coupling (between views) should be adopted. It has been introduced for modules in Structured Analysis and is explained for classes in [32] among others.

The extensive decoupling of the views on an object allows to think of *views as independent entities* which are connected by well defined relationships and constitute the object. An object then consists of a set of *object parts*, each one encapsulating one view on the object. One common suggestion could be that an object contains three object parts for the static, the dynamic, and the functional view on that object. As a first step in this direction, we fulfil a separation between control aspects concerning the use of an object (the *protocol*) and all other aspects not related to the observable dynamic of the object (the *functionality*), see figure 2. In a division of this kind, the object parts can

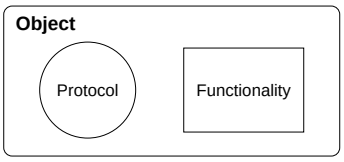


Fig. 2. Object with object parts Protocol and Functionality.

be differentiated from one another comparatively well. Besides this, the protocol is a relevant information for the clients of the object that is now explicitly stated and can be transfered outwards. It is also quite possible that additional restrictions which are not inherent in the functions are added to the protocol of an object. A further step is the exchange of the protocol so that different ways of use are possible. This topic is addressed in the example in section 4. Next, the structure of an object according to the mentioned division is presented, and the connections between protocol and functionality are explained.

3.3 Separation of Concerns

For the specification of an object in accordance with the division introduced in section 3.2, not only descriptions for the protocol and the functionality are needed, but also some for the object itself. Statecharts are well suited to the representation of protocols especially because of their compactness. Another important advantage of them is the existence of object-oriented variants. The use of statecharts to express the protocol of objects is also mentioned in [25]. In UML, a statechart is merely a view on an object in fact, while in the approach proposed in this paper it is considered as an actual part of the object. A second difference is that in UML two alternatives are provided for the processing of an operation invocation (cf. [28, p. 369]). On the one hand a special kind of event (a so-called CallEvent) can be used in the statechart of the object to handle the request, on the other hand it is possible that the object directly executes the procedure corresponding to the operation. In our approach the protocol of an object, i. e., the statechart, is the *only* one to handle operation invocations on the object. For the explanation of the connections between the different parts, a very simple statechart in form of a finite state machine is sufficient. Figure 3 shows such an automaton as a reference net; more complex statecharts can be mapped into this technique using the method described in section 2.1.4.

Not only for the functionality but also for the object itself it is difficult to find a suitable representation. Here we use reference nets, too, so that a uniform description is given for all parts and the model is executable (see figures 4 and 5). In the functionality net, the effects of the available operations on the attributes of the object are specified. The

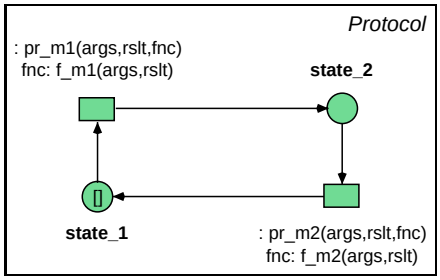


Fig. 3. The reference net *Protocol*.

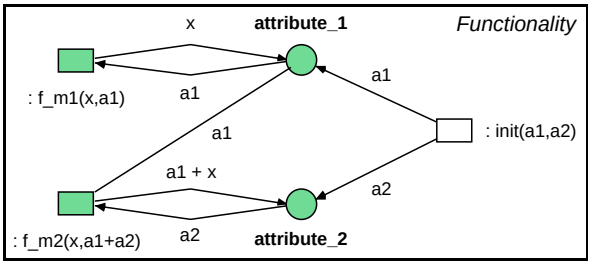


Fig. 4. The reference net *Functionality*.

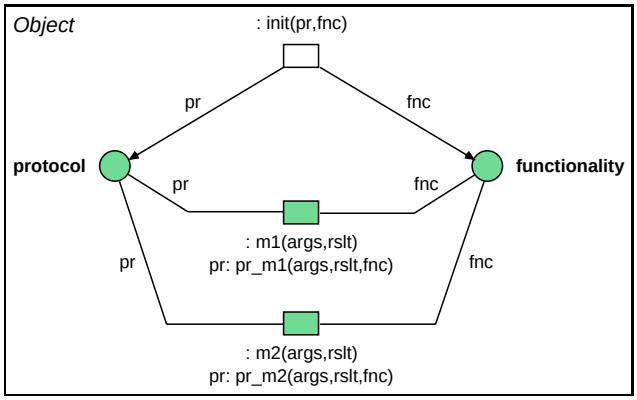


Fig. 5. The reference net *Object*.

object net contains transitions corresponding to the offered operations, but also two places, on which the references to the protocol net and the functionality net are put. Due to the representation of the protocol as a token in the object net an exchange of the protocol is eased. In the object, protocol, and functionality nets, different names for the operations are used solely for a better understanding of the connections between the nets. Only for explanation purposes the reference net in figure 6 is added. It represents the environment and encompasses a sequence of operation invocations on the object. It does not belong to the model. Among the three kinds of communication mentioned in section 3.1, only the rendezvous concept is used in the nets shown here. The synchronous channels of the reference nets are particularly suitable for this concept.

Because of the use of reference nets for all introduced parts (protocol, functionality, the object itself, and its environment), a clear semantics is given and the parts can be integrated into an overall model which can be executed

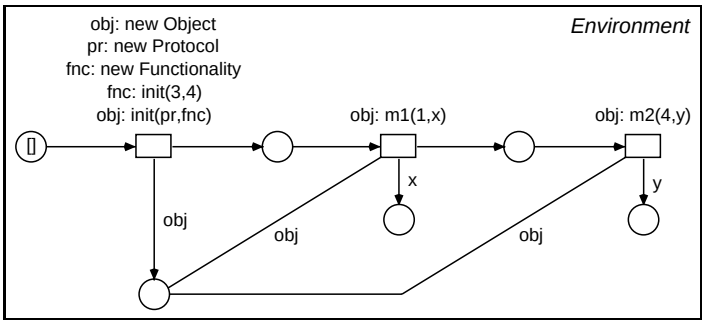


Fig. 6. The reference net *Environment*.

then. In general, it is desirable to provide comfortable representations for the parts (e. g., statecharts for the protocol) and to establish a basis for integration, execution, and verification by specifying mappings into Petri nets.

The environment net shown in figure 6 includes three transitions, which correspond to the following steps: creating and initializing protocol, functionality, and object, one invocation of operation *m1* and one of *m2*.

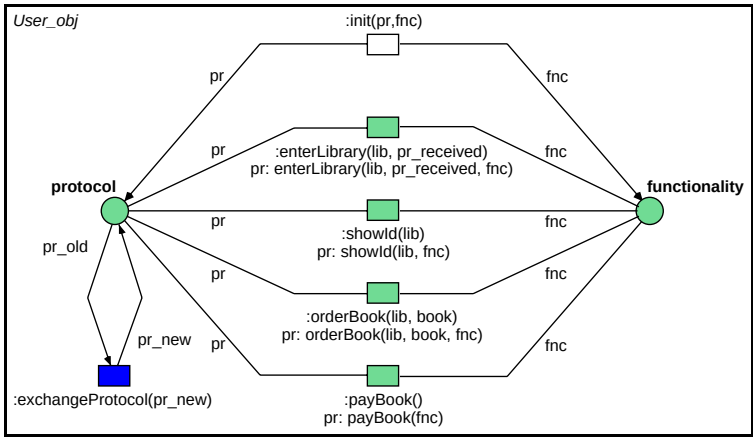
- *Step 1 (Creation and Initialization)*
  Instances of the nets *Object* (obj), *Protocol* (pr) and *Functionality* (fnc) are created. The instances fnc and obj are initialized. As a result the object has one reference to its protocol and one to its functionality. The values 3 and 4 are assigned to the attributes attribute_1 and attribute_2.

- *Step 2 (Invocation of m1)*
  There is a synchronous channel between the transition in the environment net and the upper one of the operation transitions in the object net. It leads to a unification of the labels on these two transitions, 1 with *args* (argument of operation) and *x* with *rslt* (return value of operation).
  The considered transition in the object net not only has an uplink, but also a downlink. This downlink connects it with the left transition in the protocol net by a second synchronous channel. Again, there is a unification, so the operation call is forwarded to the protocol. The additional parameter *fnc* is necessary to inform the protocol about the functionality of the object. If we assume that the protocol encompasses at least the restrictions inherent in the functionality, the protocol makes the decision on executing the functionality.
  The protocol in figure 3 allows the execution of the desired operation. By a third synchronous channel, it is connected to the transition which contains the actual functionality for the requested operation. So there exist two master-slave-relationships, one between object and protocol and one between protocol and functionality.
  Because there are tokens in all input places of the involved transitions, the transitions can fire as an *atomic* unit. As a result in the environment net, a 3 is on the place below the executed transition, the current state of the protocol is state_2, and the values of the attributes are 1 and 4, respectively.

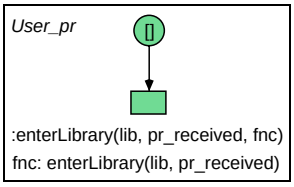
- *Step 3 (Invocation of m2)*
  The invocation of operation *m2* is processed in the same way as the one of operation *m1* in the previous step. Thereafter, the place below the last transition in the environment net contains a 5, the protocol net is in state state_1 again, and the values of the attributes are 1 and 5, respectively.

If other sequences of operations are invoked on the object, e. g. [*m2,m1*] or [*m1,m1*], a deadlock may occur. Such a deadlock is caused by the protocol which does not allow the desired sequence of operation invocations. This means that in certain states the functionality, which realizes the effects of the operations, is not executed at all. Besides deadlock there exist other possibilities for the protocol of responding to an operation invocation which is not allowed (see section 3.1). But these possibilities should not be closer inspected here.

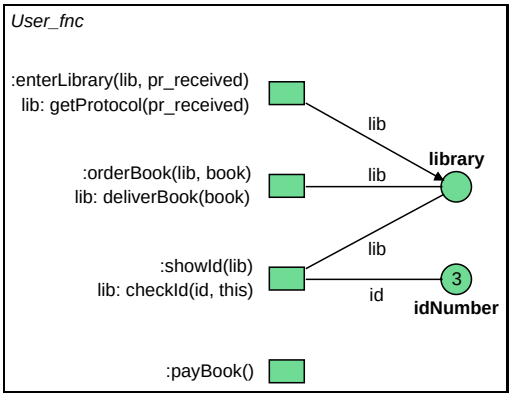
In general, a protocol of an object can have additional restrictions for the invocation of operations which are not inherent in the functionality. This is the case for the one in figure 3, for example. But a protocol should contain at least the inherent restrictions. If we consider such a minimum protocol, the protocol actually used for an object has to



(a)



(b)



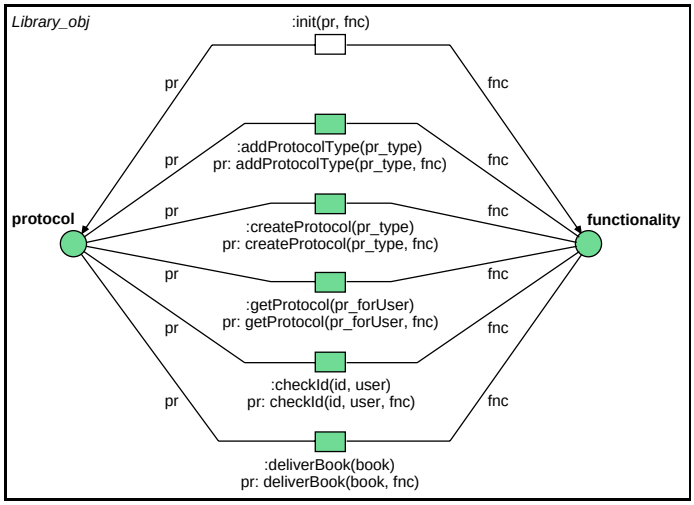
(c)

Fig. 7. Reference nets for a user object, its protocol, and its functionality.

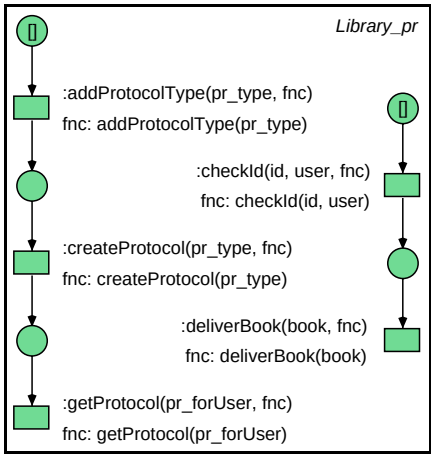
be in an inheritance relationship to it which fulfils this requirement. On the basis of Petri nets, this topic is addressed as “projection inheritance” in [1] and [2] among others. In these publications, van der Aalst and Basten also consider other kinds of inheritance for dynamic behavior . If there is an inheritance relationship between two classes, this has an important effect on the dynamic behavior of instances of these classes and therefore on their protocols. So-called “inheritance anomalies”, which can occur in this context, are addressed in [21].

4 An Example

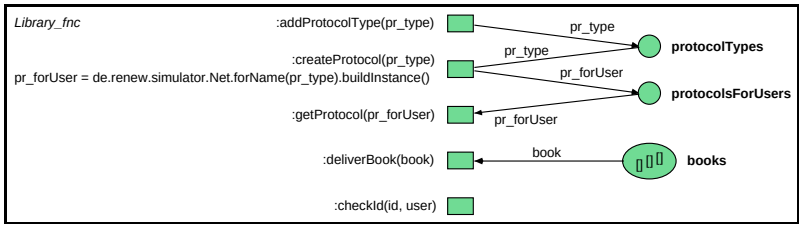
The example given in this section illustrates the ideas presented in the previous one. In addition, the main concern of the example is to present communication between objects and exchanging the protocol of an object. It considers a segment of a library system. Two objects are introduced: object *user* and object *library*. As it is defined in section 3.3, each object can be represented by three nets which stand for its protocol, its functionality and the object itself. In figures 7(a)–7(c) the object *user* is represented, respectively the object itself, its protocol, and its functionality. In figures 8(a)–8(c) the object *library* is represented the same way.



(a)

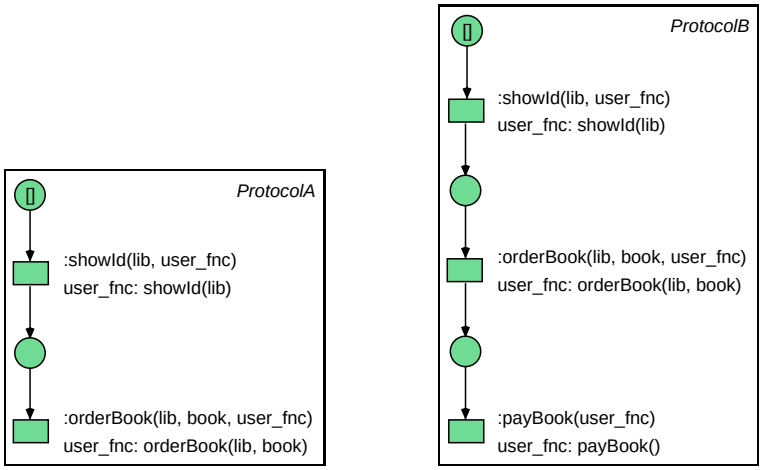


(b)



(c)

Fig. 8. Reference nets for a library object, its protocol, and its functionality.



(a) (b)

Fig. 9. Different protocols for a user.

When the user enters the library, he receives information about actions that he has to perform in order, for example, to borrow a book. This information is represented as a *protocol*. Also, the user can go into different libraries, and his behavior has to depend on the specific library. Therefore, when the user enters a library, his initial protocol has to be changed. The user behaves according to the rules of the specific library represented by a corresponding net.

In figures 9(a) and 9(b), two different nets are shown representing two different sequences of actions the user has to follow in order to borrow a book in two different libraries. For the case of simplicity, the book involved is represented as a black token. Additional assumptions are made. It is supposed that the user's functionality encompasses all actions that he can or has to perform for at least one library.

Figure 10 shows a reference net that represents an application logic. Its initial marking allows firing a transition with several inscriptions. As a result, two objects are created: objects *user* and *library*, which are put in places user and library, respectively. The next transition of the net *Environment_Lib* adds a special protocol type to the library, namely "ProtocolA" (figure 9(a)). After this step, a concrete instance of this protocol type is created inside of the library.

The reference net for object *user* is represented in figure 7(a). Places protocol and functionality contain the user's protocol and functionality, respectively. The reference nets of the user's protocol and functionality are represented in figures 7(b) and 7(c). The user's identification number is given in form of an integer in the place idNumber of the reference net representing his functionality. His initial protocol is a simple net with one transition. Firing this transition results in entering the library and receiving a new protocol from it. The actual change of the user's protocol is performed by firing the transition with inscription `:exchangeProtocol(pr_new)` (figure 7(a)). This service has one parameter which denotes a reference to the net representing the protocol of the library. Then, the user begins to behave according to the rules of the specific library. The new protocol represents the sequence of actions the user has to perform in order to borrow a book. Figure 9(b) shows a further protocol, that has one additional transition with inscription `:payBook(user_fnc)` and can be handed over to the user in another library.

The reference net for the *library* object is represented in figure 8(a). Its places protocol and functionality contain the library's protocol and functionality, respectively. The corresponding reference nets are shown in figures 8(b) and 8(c). As quoted above, a book is represented as a black token in figure 8(c). Several unnamed places are shown in figure 10 in order to visualize control flow. Objects *user* and *library* communicate with each other by invoking services through synchronous channels.

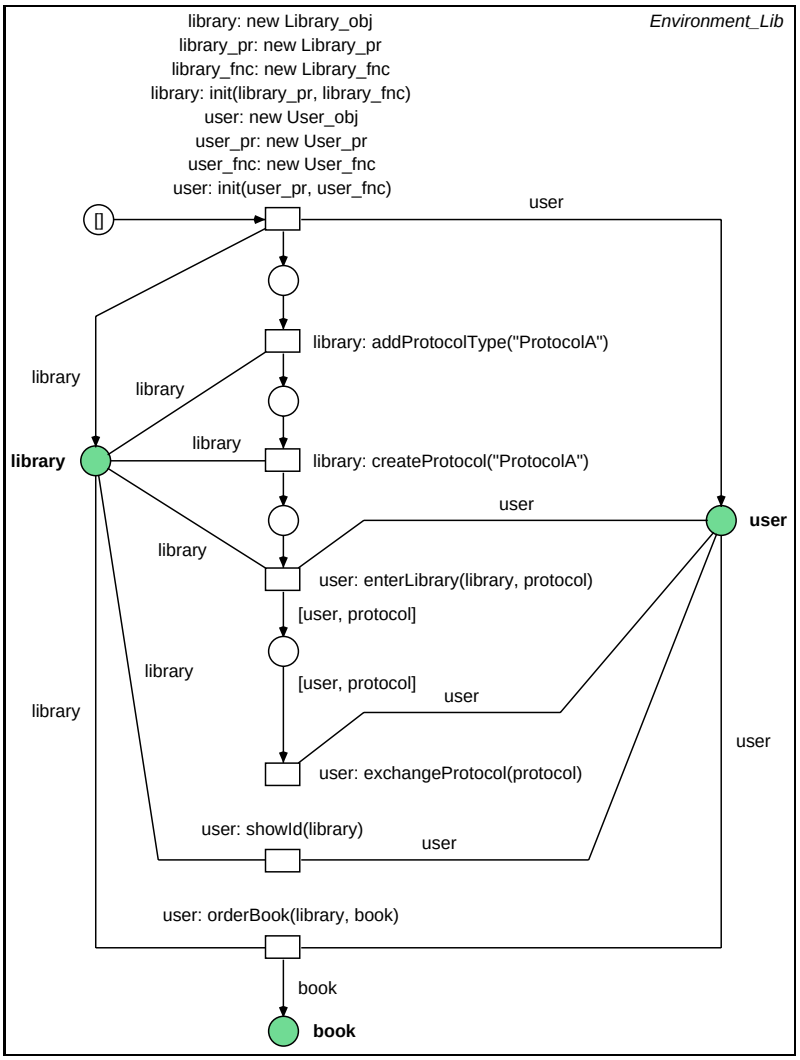


Fig. 10. The reference net *Environment_Lib*.

5 Conclusion

In order to improve the specification of object-oriented systems and thereby to ease the transition from specification to implementation, a model of objects has been proposed which concentrates on views on objects. An object is thought of as consisting of several parts that encapsulate the views on it. As a first step in this direction, the protocol of a class has been extracted and represented as a statechart here. The integration of objects and their parts has been illustrated on the basis of reference nets. This visualization shows that specifications can become clearer due to the separation of the protocol and the functional part. Additionally, protocol-related aspects can be changed more easily during the specification process. In the example it is demonstrated that the adaptation of objects according to new environments can be established in a straightforward manner, namely via exchange of protocol. However, the model proposed here results in larger objects.

In the future, several topics will be addressed. It has to be clarified in detail how the elements and features of statecharts can be used in order to define protocols. An investigation of the connections between effects of operations on attributes on the one hand and state changes in the statechart on the other hand is necessary, too. Another important

question deals with exchanging of protocols: it has to be cleared how active states of a statechart an object has just received as protocol can be deduced from active states of its old statechart. A further aspect is the consideration of processes involving several objects, e. g., workflows, and their integration with the object descriptions provided in our approach.

References

[1] van der Aalst WMP, Basten T. Life-Cycle Inheritance: A Petri-Net-Based Approach. In: Azéma P, Balbo G (eds) Proceedings of the 18th International Conference on Application and Theory of Petri Nets. Springer-Verlag, Berlin, Heidelberg, New York, 1997, pp 62-81 (Lecture Notes in Computer Science No. 1248)

[2] Basten T. In Terms of Nets: System Design with Petri Nets and Process Algebra. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1998

[3] von der Beeck M. A Comparison of Statecharts Variants. In: Langmaack H, de Roever WP, Vytopil J (eds) Formal Techniques in Real-Time and Fault-Tolerant Systems. Springer-Verlag, Berlin, Heidelberg, New York, 1994, pp 128-148 (Lecture Notes in Computer Science No. 863)

[4] Breu R, Grosu R. Modeling the Dynamic Behavior of Objects on Events, Messages and Methods. Technical Report TUM-I9804, Institut für Informatik, Technische Universität München, 80290 München, Germany, 1998

[5] Christensen S, Damgaard Hansen N. Coloured Petri Nets Extended with Channels for Synchronous Communication. Technical Report DAIMI PB - 390, Aarhus University, Computer Science Department, DK - 8000 Aarhus C, Denmark, 1992

[6] Coleman D, Hayes F, and Bear S. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. IEEE Transactions on Software Engineering 1992; 18(1):9-18

[7] Douglass BP, Harel D, Trakhtenbrot M. Statecharts in Use: Structured Analysis and Object-Orientation. In: Rozenberg G, Vaandrager F (eds) Lectures on Embedded Systems: European Educational Forum School on Embedded Systems. Springer-Verlag, Berlin, Heidelberg, New York, 1998, pp 368-394 (Lecture Notes in Computer Science No. 1494)

[8] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts 01867, 1995

[9] Guttag J. Abstract Data Types and the Development of Data Structures. Communications of the ACM 1977; 20(6):396-404

[10] Harel D. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 1987; 8:231-274

[11] Harel D, Gery E. Executable Object Modeling with Statecharts. In: Proceedings of the 18th International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996, pp 246-257

[12] Harel D, Naamad A. The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology 1996; 5(4):293-333

[13] Harel D, Pnueli A, Schmidt JP, Sherman R. On the Formal Semantics of Statecharts. In: Proceedings of the Symposium on Logic in Computer Science. IEEE Computer Society Press, Washington, DC, USA, 1987, pp 54-64

[14] Harel D, Politi M. Modeling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw-Hill, New York, NY, 1998

[15] Jensen K. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1. Springer-Verlag, Berlin, Heidelberg, New York, 1992

[16] Kay AC. The Early History Of Smalltalk. ACM SIGPLAN Notices 1993; 28(3):69-95

[17] Kummer O. Simulating Synchronous Channels and Net Instances. In: Desel J, Kemper P, Kindler E, Oberweis A (eds) 5. Workshop Algorithmen und Werkzeuge für Petrinetze. Forschungsbericht Nr. 694, Universität Dortmund, Fachbereich Informatik, Dortmund, Germany, 1998, pp 73-78

[18] Kummer O, Wienberg F. Renew — User Guide, Release 1.0. University of Hamburg, Department for Informatics, 22527 Hamburg, Germany, 1999 (URL: http://www.renew.de)

[19] Liskov B, Zilles S. Programming With Abstract Data Types. ACM SIGPLAN Notices 1974; 9(4):50-59

[20] Martin J, McClure C. Diagramming Techniques for Analysts and Programmers. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1985

[21] Matsuoka S, Yonezawa A. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In: Agha G, Wegner P, Yonezawa A (eds) Research Directions in Concurrent Object-Oriented Programming. The MIT Press, Cambridge, Massachusetts; London, England, 1993, chapter 4

[22] Meyer B. Eiffel: The Language. Prentice Hall International (UK) Ltd, Hertforshire HP2 4RG, 1992

[23] Meyer B. Object-Oriented Software Construction. Prentice Hall PTR, Upper Saddle River, New Jersey 07458, second edition, 1997

[24] Moldt D. Höhere Petrinetze als Grundlage für Systemspezifikationen. PhD thesis, Universität Hamburg, Hamburg, Germany, 1996

[25] OMG Unified Modeling Language Specification, Version 1.3. 1999 (URL: http://www.omg.org/cgi-bin/doc?ad/99-06-08)

[26] Pnueli A, Shalev M. What is in a Step: On the Semantics of Statecharts. In: Ito T, Meyer AR (eds) Theoretical Aspects of Computer Software. Springer-Verlag, Berlin, Heidelberg, New York, 1991, pp 244-264 (Lecture Notes in Computer Science No. 526)

[27] Rumbaugh J, Blaha M, Premerlani W, Eddy F, and Lorensen W. Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991

[28] Rumbaugh J, Jacobson I, Booch G. The Unified Modeling Language Reference Manual. Addison-Wesley, Reading, Massachusetts 01867, 1999

[29] Tanenbaum AS. Computer Networks. Prentice-Hall, Upper Saddle River, New Jersey 07458, third edition, 1996

[30] Valk R. Modelling of Task Flow in Systems of Functional Units. Technical Report FBI-HH-B-124/87, Universität Hamburg, 22527 Hamburg, Germany, 1987

[31] Valk R. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: Desel J, Silva M (eds) Proceedings of the 19th International Conference on Application and Theory of Petri Nets. Springer-Verlag, Berlin, Heidelberg, New York, 1998, pp 1-25 (Lecture Notes in Computer Science No. 1420)

[32] Yourdon E, Argila C. Case Studies in Object-Oriented Analysis and Design. Prentice Hall PTR, Upper Saddle River, NJ 07458, 1996