A Petri Net Approach to Conceptual Modelling and CASE

Thomas Marx [a]

[a] *Department of Computer Science, University of Koblenz-Landau, Germany, email: tom@uni-koblenz.de*

Abstract

We present a Petri net based approach for the development of information systems. By the use of diagram techniques software development is made transparent to the system engineers and users. The initially informal descriptions become more and more specified and formalised. Seamless, the same diagrams are used throughout all phases. Synthesising common techniques and adjusting them to each other Petri nets are central; they offer means to simulate and analyse system's behaviour as well as the generation of prototypes is established. Here, the CASE-tool NEPTUN supports a platform and database system independent generation, decoupled from the system's generic model. In this paper we sketch the main objectives of our approach and give some more information on requirements-engineering.

Keywords: CASE, Petri nets, prototyping, information systems, requirements engineering

1 Introduction

The design and the development of software systems are complex tasks. There are three major problems: At the *extent* of a project that is often too large for one or a small group of developers, at the *innovation* of activities since system development includes only a small degree of routine, and at the great number of the *dependencies* stratified on several tiers between the different developers and fields.

Traditionally in CASE a three layered architecture is offered to face these problems: The *method layer* that arranges the overall process and provides a systematic relief, the *formal layer* that aims at an automatic transformation of the system specification, and the *tool layer* that supports the developer's work.

NetCASE is used for the method layer, augmenting traditional waterfall models. For the formal layer Petri nets and Object Model Diagrams (OMDs) are combined introducing special PrT-net subclasses and for the tool layer NEPTUN offers means for editing, analysing and simulating diagrams, including automatic generation of prototypes.

Today mostly waterfall-like *phasemodels* are used for the method layer, giving the emphases of the different phases. The first phase is necessarily occupied by the requirements definition. It is important, to specify the system demands of the user at the beginning of software development correctly, precisely and clearly. Already small mistakes and inaccuracies have immense effects on the further phases; implications on costs and time are the largest ones concerning the whole process of development. This shows the importance of an appropriate support at requirements engineering and a seamless take over for subsequent phases. This and the possibility to easily step back at any time is lacking very often in other approaches.

In order to support the developer and to visualise the results, diagram techniques become today's allied for all three layers. The advantage is the immediate use of man's visual perception, supporting a direct and simultaneous access to different system components. Plain text on the other hand is organised sequentially, reading is trained. Therefore diagram techniques are favoured here, synthesising known ones and avoiding redundant concepts. Only in order to offer better support for requirements engineering a new approach, based on Petri nets, was established.

Many CASE methods and tools employ rapidly new (technical) concepts and ideas; on the other hand there are only few means or improvements to support the dialogue between developer and client. Most times a *summative evaluation* of the system is done. A *formative evaluation*, that supports the tuning of system extracts and aspects according to the user's demands, is rare. Hence our approach tries to offer means for formative evaluation as soon as possible.

Today's approaches in CASE are designed or at least propagated to be universal. A wide as possible spectrum of software systems is aimed at. Individual demands of the domain are simply treated in a general way. In NetCASE we restrict ourselves to the development of information systems, offering good support for declarative access to large amounts of data, transparent to the underlying database system. Additionally the visual system representation can be adjusted to the specific needs. Nevertheless the arbitrary restriction does not imply that no other kinds of software systems besides information systems can be engineered.

In this paper we give a brief overview on our modelling approach. Additionally a deeper insight into requirements engineering is given.

2 NetCASE

NetCASE [Marx, 1995] supplies a phasemodel, guiding the process of development, as well as notations and formalisms in order to model system facilities.

The four stages: Requirements, design, implementation model and software product illustrate the change of development's emphasis and objectives. Here, the viewpoint shifts from a process-oriented one to an object oriented one, finally being concerned with the product itself.

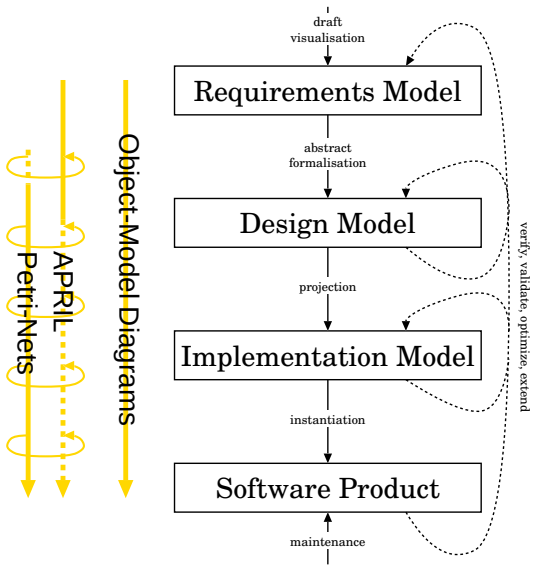

Figure 1. phasemodell

Figure 1 presents a coarse view on NetCASE. On the left the emphases of the three different notations is given. Introduced once, a notation is employed in all following steps up to shipping the product. This supports recourse at any time for possible corrections and repairs; the immediate use of preceding results and activities is given, too. Methodical gaps, additional transformation, rewriting and misunderstandings are avoided from the scratch.

Diagrams often include icons and symbols. Icons and symbols make good use of empirical and common knowledge, improving the insight and the decrease of decoding time for the observer. Clear representations may become metaphors of the states of affairs. Here, a reasonable approach is to synthesise the *representation of the abstract algorithm* [Brown, 1988] and the *animation of programmes* into *software visualisation* [Domingue *et al.*, 1994]. Software visualisation is used in NetCASE.

Since the different stages of software development address different demands of abstraction, intuition, completeness and so forth to the representation form, two viewpoints are provided in NetCASE: APRIL diagrams and Petri nets. On different levels, both serve for modelling functional and dynamic system behaviour, where orthogonal OMDs are used for the more static view on data. The elements of APRIL diagrams and Petri nets are mapped on each other.

In requirements engineering the protruding role is played by APRIL diagrams, while in the progress of software development they resign behind Petri nets without loosing their importance or intuition; viewpoints may toggle at any time. When requirements engineering is accomplished and Petri nets are

added. Consistency between both views is assured immediately while editing.

The motivation to provide different viewpoints to the state of affairs is based on the necessity to offer a less abstract language for system design including the frankness for incomplete specifications [Kop/Mayer, 1996]. The optimal use of results is warranted in NetCASE by the conceptual integration of Petri nets and APRIL diagrams. Quality improvements are achieved by *validating* the user's requirements and by *verification*, in the sense of consistence and simulation, both for the user and for the developer.

It is helped by the domain dependent adaptability of descriptions to detect major and minor problems; the application field is put in the centre of discourse. This is in contrast to the static and generalised environments that are encountered very often. The APRIL representations are arranged according to the processes of work respectively to the integration of the software system into them. By modelling these processes the domain dependent objects and classes are identified and become modelled by OMDs.

Consequently NetCASE sells itself more as a process oriented [Moehring, 1994] than an object oriented approach, in particular concerning requirements engineering. However, object oriented concepts are integrated and especially for modelling static affairs.

3 Petri nets

In NetCASE Petri nets are fundamental for design and all further phases of development. They provide a complete description of the system. Two subclasses of Predicate-Transition nets (PrT-nets) [Genrich/Lautenbach, 1981] are used in NetCASE: *data processing nets (DPNs)*, used for modelling causal relationships between actions with regard to control and data flow, and *system navigation nets (SNNs)*, used for modelling user interaction in the scope of system navigation; note that this can not be compared with terms of navigation from database theory. Only DPNs are used for integrating object models.

The available means for system navigation by users are given by SNNs. DPNs possibly contain user interaction, too, but they are basically intended for data access. SNNs are set up for a better distinction and to expose status. Compared to other approaches they are related to use cases [Booch, 1994] and interaction diagrams [Jacobsen, 1992]. SNNs are kept pretty simple in order to stress the main concepts and to avoid misuse. They are founded on *condition event nets (ce-nets)* [Reisig, 1982], a subclass of PrT-nets. Thus, places and edges are marked with one anonymous token only.

Directed by control flow and restricted by expressions, flow of and access to data is modelled by DPNs. This inherent combination avoids the lack of inter-

operability, found in many methods for conceptual modelling. DPNs illustrate sequences and interdependencies of system states respectively to system dynamics; note that basic functionality and static information is annotated and still captured local to classes in line with the object-oriented paradigm of encapsulation. In order to relate DPNs to OMDs, places are identified with classes; any other places are used for modelling controlflow. The extent of the class becomes the marking of the place. Whenever a transition fires that is connected by an incident arc the extent of the class is affected. For more information on SNNs and DPNs see [Marx, 1995].

Petri nets are used for simulation and prototyping. Here, system behaviour can be visualised simultaneously by the token flow in Petri nets as well as by the object flow in APRIL diagrams. Validation and evaluation are accomplished with regard to the initial requirements model.

The generated prototype includes database bindings, graphical user interfaces and sockets to NEPTUN [Simon, 1996]. Latter enhances the prototype by means of APRIL diagrams and Petri nets at the same time. Unlike animation no additional expenditure results for simulation, if the Petri net based design is accomplished; neither for preparation nor for maintenance. Simulation is the direct model illustration, running a prototype simultaneously. Here, Petri nets offer a deep insight into the system's model as well as means for immediate correction.

4 Object Modelling Diagrams

The modelling of classes, objects, local behaviour and so forth is accomplished in NetCASE by object model diagrams (OMDs), according to the Object Modelling Technique [Rumbaugh *et al.*, 1991].

In OMDs the classes are defined by attributes, derived attributes, methods, aggregation, inheritance and associations to other classes. The attribute types can be self-defined via set, list and tupel constructor, based upon a set of standard types. Standard integrity constraints can be annotated and a sophisticated integrity check mechanism [Fahrner *et al.*, 1995] secures the declarative formulation and efficient checking, making good use of inheritance.

Whenever the database scheme is generated, some concepts become simulated depending on the scope of the target system. Some things like multiple inheritance, n-ary relationships, constraints between associations and some minor ones are not supported, because there is no real need for and/or it would be hard to simulate them by some database management systems.

In OMDs it is given whether a class is transient or to which persistent logical database it belongs to. This is required, because distributed heterogeneous

databases and applications can be generated. Here, the logical databases are instantiated by possibly different ones, residing on possibly different hosts. Currently only Informix, a relational database system, and ObjectStore, an object oriented one, bindings are exemplary established.

5 APRIL

Providing appropriate representation for objects, actions, processes and interdependencies, APRIL [Marx, 1997] supports real communication between user and system developer. The focus is on an intuitive visualisation for both sides, while depths and completeness may vary. APRIL can be adapted to the particular demands of the individual system development. Step-by-step APRIL drafts are patched and refined together by the user and system developer. They are used during the whole process of development, providing an abstract and illustrative view.

Traditionally prototypes and visualisation of system dynamic's are only available at the end of system engineering. In order to illustrate the system's future dynamics very early, APRIL diagrams provide animation. Because separate parts can be animated, a premature possibility for formative evaluation is given.

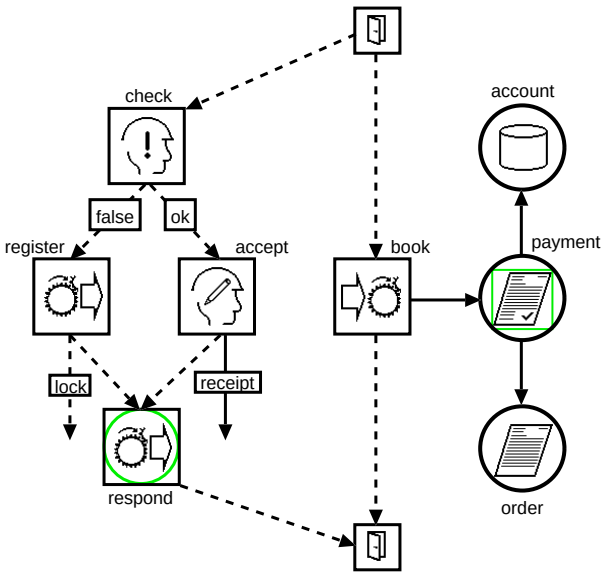

Figure 2. APRIL diagram

APRIL diagrams become specified more precisely by Petri nets providing their formal semantics. Therefore APRIL diagrams remind of Petri nets. Static aspects are modelled by OMDs, associated with APRIL diagrams and Petri nets. In the following the basic elements, animation and transformation of APRIL diagrams is presented.

5.1 Basic elements

Objects and objectstores, given by circles, are distinguished and assigned to OMD classes. As a result, the static information is directly available for connotations in APRIL by identifying objectstores with classes; however, there are no OMD representatives for objects. Objects themselves are not given

in APRIL diagrams, but they are manifested first at animation. The basic characterisation of objects is given by the corresponding class.

Different kinds of actions are denoted by rectangles, analogously to transitions in Petri nets. A process is a set of actions that are in causal dependencies, combined into an APRIL diagram here. This corresponds to the concept of procedures, offering means for hierarchy formation. Any process is associated to exactly one APRIL diagram and is denoted by doubled rectangles, not given in Figure 2. Values of subprocesses are returned by dangling dataflows (`receipt`).

A solid arrow represents a dataflow of a set of objects. Four different kinds of dataflow edges respectively combinations of them are distinguished: *production*, *selection*, *modification* and *consumption* of objects. Only one kind of access is allowed for each action, because otherwise a compound access would lead to an inaccurate semantic. In APRIL dataflows are at least at one side combined with an objectstore. They are the exclusive link between actions and objectstores. For those dataflows that run between two objectstores it is reasonable to indicate the active character of one of the objectstores by an inscribed rectangle (`payment`). No genuine controlflow is modelled here; `account` and `order` are accessed implicitly.

The system status is characterised by those actions that are active. Dashed arrows represent controlflow, only taking place between actions. Control is passed on from action to action where concurrencies and alternatives can be mounted. Latter are synchronised respectively reassembled at least at the end of each process in order to provide well defined start and end points.

Actions do not act autonomously in APRIL, but become activated by terminating predecessors. Processes can multiply fork into an arbitrary set of concurrent ones. Consequently APRIL provides a hybrid execution model; it is neither distributed nor centralised. The communication of concurrent processes is purely in asynchronous mode by the use of common data. Transaction mechanisms secure the correctness of data operations.

Concurrency is introduced by several non-inscribed outgoing edges of an action. Whenever synchronising threads all predecessors have to terminate. Outgoing labeled/conditioned edges (`check`) represent alternatives. E.g. there are alternatives reassembled at Action `respond`. Therefore a shaded circle is inscribed to indicate its non-synchronising character. The alternative's junctions are already activated by the termination of any predecessor. This corresponds to the semantics of the underlying Petri net: the transition becomes activated by the marking of its internal place.

Launching and receiving events is denoted by *event edges* (`lock`), represented in APRIL by dashed arrows that hang loose at one side. Events start new

concurrent threats by involving APRIL diagrams. Basically events can be simulated by involving subprocesses concurrently. In this case the subprocess must distinguish which incoming event edge got triggered and/or which was the parent process. The event concept offers a more convenient mechanism here.

5.2 Animation

Animation is based on the informal description of APRIL diagrams which is complemented manually by the developer around animation flows. Latter are assigned to APRIL diagrams and can be embedded into each other via processes. Animation is consequently predetermined by the developer. The degree of interaction is hardcoded into each course.

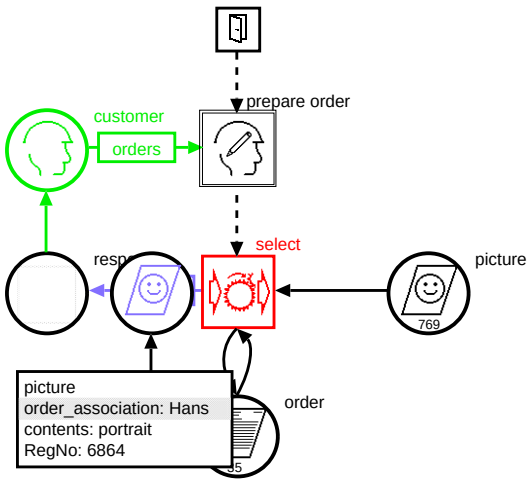

Figure 3. animation

An animation illustrates the temporal sequence and interdependencies of actions. Since modelling by APRIL is based on a process oriented view, this is an important mean for verification and validation. The impression of an early prototype is provided. Here, developer and client work constructively together. Because of the lacking depth and accuracy of APRIL diagrams, it devolves on the developer to build animation flows manually. This offers the possibility to emphasise important system components explicitly.

Figure 3 shows animation in APRIL. `Pictures` are consumed by `select` and new `response` objects are generated. In this case, the active action is dyed red and the current dataflow is dyed blue. The modified attributes, the association between `response` and `order` here, are shaded. The number of objects per objectstore is given numerically; icons are missing at empty objectstores. The flow of objects along the edges is animated graphically and the object's attributes and associations are presented.

In the progress of software development, the investigation of system dynamics by animation is replaced gradually by simulation. Here, a Petri net view is offered in addition. The system behaviour becomes directly illustrated, while the developer is released from additional work by Petri nets, because simulation is purely based on Petri nets descriptions.

5.3 Transformation

The transformation and mapping of APRIL diagrams on Petri nets takes place entering the system design phase and is divided into three steps: *correction*, *completion* and *conversion*. By conversion the APRIL elements are finally mapped on net elements so that two different views are available.

In the correction phase, the APRIL diagrams are checked for plausibility. Erroneous aspects concerning alternatives, concurrencies or ambiguity are discovered. The defects are presented to the developer and correction is suggested whenever it can not be accomplished automatically. In case of complex correction transformation terminates. The changes are finally made persistent.

Completion is fully transparent to the developer. Those nodes, edges and refinements are included that are implicit to complex elements like events, active objectstores and so on. Missing edges to the end action are added as well as loops are resolved. Since APRIL elements obviously imply these complements, they are not made persistent. They built the bridge to convert APRIL into Petri nets.

The conversion essentially includes the generation of Petri nets, based on completed APRIL diagrams. Here, the mapping of Petri net and APRIL elements is stipulated. Conversion occurs fundamentally canonical by complementing the missing controlflow places. Furthermore, complex actions are split into transition sequences. A complex action is one where various edges are mounted.

In a *stable* NetCASE system model every APRIL edge is associated to a Petri net edge, every objectstore to an non empty set of data stores and every action to a transition-encapsulated subnet.

6 NEPTUN

The CASE-tool NEPTUN supports the editing, analysis, generation and simulation of diagrams. Associations between elements of different diagrams are made on heuristics or by hand; they can be visualised and modified explicitly at any time.

Concerning requirements engineering the animation editor's functions include generation, processing, deletion and migration of objects as well as the sequential, optional and concurrent propagation of controlflow; refinements can be involved, too. Consequently an animation flow is a sequence of actions that operate on incident objectstores. The extent of the objectstores is modelled exemplary while these instantiated objects can be accessed by actions. Here, the individual attributes of every object can be determined. An animation browser is provided to pursuit and to organise the controlflow. Refinements

are embedded via local animation flows; embedded into an APRIL diagram, any animation flow of a subprocess can get involved. In contrast to animation the simulation interface is far more simple. Except for contolflow handling and direct data access no additional functionality must be provided.

Besides CASE functionality the adaptable and balanced design of APRIL diagrams is a core concept for an appropriate relief for conceptual modelling, especially for requirements engineering. In APRIL nodes are provided with icons. A hierarchy of predefined icons is given initially. A small part of the hierarchy is given in figure 4. These icons ease the first use of APRIL and form a guideline for successive augmentation and modification.

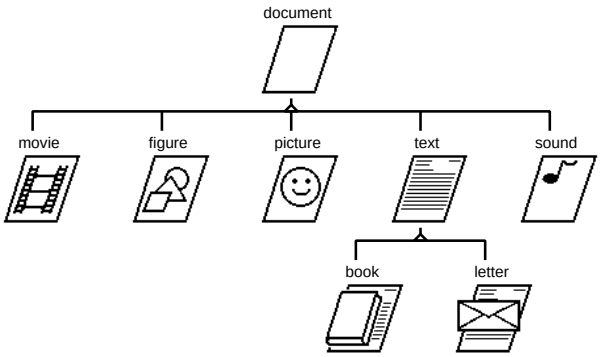

Figure 4. icon tree (extract)

Icons need not to be studied. By a homogenous and good layout they support the recogniseability, differentiation and nominal value [Foley *et al.*, 1990]. Compared to textual elements their semantic field is wider. The provided iconography, the lexicon of the icons in APRIL, is organised by terms like `process`, `event`, `object`, `document`, `book`, `text` and so on. The use of icons avoids misunderstandings on account of subjective or cultural standards and experiences as well as a rapid perception is supported. The icons in APRIL are reduced to the object's or the activity's essential qualities. For additional categorisation and extra determination, representations can be combined and/or extended. In such way, `document` is refined by `text`, and `text` is refined by `letter` and `book`. Additional labels are used for distinction and determination. The homogenous node size and line thickness as well as the limitation to a fixed kind of nodes (two) and edges (four) support diagram's simplicity. Additional dimensions are not required, but may lead to a superfluous visual load.

APRIL's design is based on the experiences of successful diagram designs for heterogeneous groups of persons, as is for instance the London underground plan [Mullet/Sano, 1995]. A rapid perception for heterogeneous groups is achieved by homogeneous design features, good style and by sacrificing exact representation of lengths and ratios. The compliance of style and the formulation of design guidelines make APRIL become an appropriate and profitable mean for communication [Schweitzer, 1997].

There are some remarkable technical features of NEPTUN. On account of the

broad availability of the virtual engine, the generated Java prototypes run on many platforms. By the use of AWT-classes[Flanagan, 1996] they offer a graphical user interface. The underlying database can be chosen at generation, e.g. Informix or ObjectStore, including the optional generation of the data scheme. Since the prototypes use generic database interfaces, the database management system can be replaced even for already generated and running applications.

NEPTUN itself was developed at the University of Koblenz-Landau in C++ and runs on solaris/x-windows. A win32 version is under construction.

7 Outlook

Today NetCASE was only applied to small examples so that real world applications will be tested in future works. The experiences may imply corrections and extensions.

It is obvious that the representation of transactions and their boundaries become more important so that the formulation of transactions in APRIL diagrams will be necessary. Therefore object oriented concepts for complex integrity constraint will be added. Constraints for alternatives in APRIL will be relaxed by making conditions local to each controlflow.

Currently a module for reengineering Oracle databases and applications is added. Here, schema information from a database and its applications, possibly hardcoded before, is retrieved, visualised and imported into NEPTUN.

The integration of Petri nets and APRIL diagrams for simulation is yet quite rudimentary and will be more elaborated. Additionally some more features will be added to NEPTUN, including the increase of predefined icons, a more comfortable animation editor, checks for model's feasibility and so on. A win32 version will be provided, too. This will make NEPTUN even more attractive for software engineering.

References

[Booch, 1994]
Grady Booch. *Object-Oriented Analysis and Design.* Benjamin/Cummings, 2. Auflage, 1994.

[Brown, 1988] **M. H. Brown**. Algorithm animation. Technical report, MIT Press, 1988.

[Domingue *et al.*, 1994] **J. Domingue, B. A. Price, M. Eisenstadt**. Viz: A framework for describing and implementing software visualization systems.

In **D. J. Gilmore, R.L. Winder, F. Detienne**, (Hrsg.), *User-Centered Requirements Engineering Environments*. Springer Verlag, 1994.

[Fahrner *et al.*, 1995] **Christian Fahrner, Thomas Marx, Stephan Philippi**. Integration of integrity constraints into object-oriented database schema according to odmg-93. Technical Report 9, University of Koblenz-Landau, 1995.

[Flanagan, 1996] **David Flanagan**. *Java in a Nutshell*. O'Reilly & Associates, 1996.

[Foley *et al.*, 1990] **James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes**. *Computer graphics: principles and practise*. Addison Wesley, 1990.

[Genrich/Lautenbach, 1981] **Hartmann J. Genrich, Kurt Lautenbach**. System modelling with high-level petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[Jacobsen, 1992] **Ivar Jacobsen**. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[Kop/Mayer, 1996] **Christian Kop, Heinrich C. Mayer**. Objektorientierte Analyse und konzeptueller Vorentwurf. *EMISA Forum*, 1, 1996.

[Marx, 1995] **Thomas Marx**. NetCASE - A Petri Net based Method for Database Application Design and Generation. *Fachberichte Informatik, University of Koblenz-Landau*, 11/95, 1995.

[Marx, 1997] **Thomas Marx**. APRIL - Visualisierung der Anforderungen. Technical Report 7, University of Koblenz-Landau, 1997.

[Moehring, 1994] **Michael Moehring**. Grundlagen der Prozessmodellierung. Technical report, University of Koblenz-Landau, 1994.

[Mullet/Sano, 1995] **Steve Mullet, Darell Sano**. *Designing Visual Interfaces*. Prentice Hall, 1995.

[Reisig, 1982] **W. Reisig**. *Petri-Netze (in German)*. Springer-Verlag, 1982.

[Rumbaugh *et al.*, 1991] **James Rumbaugh, Michael Blaha, Wiliam Premerlani, Frederick Eddy, William Lorensen**. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1. Auflage, 1991.

[Schweitzer, 1997] **Thomas Schweitzer**. APRIL - Animierter Prozessillustrator. Master's thesis, University of Koblenz-Landau, 1997.

[Simon, 1996] **Carlo Simon**. Programmieren mit Netz-Spezifikationen. Master's thesis, University of Koblenz-Landau, 1996.